The McGraw-Hill Companies

Grid Computing
A Research Monograph
(Book + CD)

Grid Computing
A Research Monograph
(Book + CD)

D. Janakiram

Distributed & Object Systems Lab,
Department of Computer Science and Engineering
Indian Institute of Technology
Madras

Tata McGraw-Hill Publishing Company Limited

NEW DELHI

McGraw-Hill Offices

New Delhi New York St Louis San Francisco Auckland Bogotá
Caracas Kuala Lumpur Lisbon London Madrid Mexico City Milan
Montreal San Juan Santiago Singapore Sydney Tokyo Toronto

Tata McGraw-Hill

ISBN 0-07-060096-1

Published by the Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008, typeset in Baskerville BE Regular
at The Composers, 260, C.A. Apt., Paschim Vihar, New Delhi 110 063 and
printed at Ram Book Binding, C-114, Okhla Industrial Area, Phase 1,
New Delh110 020

Cover: De-Unique

RAACRDRBRYQCY

Preface

Grid Computing is a monograph depicting, in a chronological order, the evolution and progress of models of computation, from clusters to grid computing.

In the early 90s, clusters of workstations were popular in the academic computing scenario. The attempt to optimally use them beyond standalone workstations propelled the beginnings of models that used workstations for parallel computing. The interesting evolution of these models over the past 10 years at the Distributed and Object Systems Lab, IIT, Madras, forms the subject of this monograph. It depicts the growth of models for parallel computations on workstations from cluster computing to grid computing in general.

The initial efforts, which began in the early 90s, focused on proving the concept of mixing parallel loads with sequential loads on the same network of computing systems. The challenges included operating system, hardware heterogeneities, differences in computing speeds of the machines, and load imbalances. Various models were proposed over the years to achieve this goal. These programming language constructs, designed for parallel programming, are simple and can be used to teach the designing and implementation of distributed languages. The performance studies bring out the tradeoffs between providing flexibility to the programmer versus efficiently implementing the constructs at run-time.

Anonymous Remote Computing (ARC) is an attempt at cleanly separating the programmer's concerns from the system's concern.

The programmer only attempts to express the maximum parallelism available in the program. It is left to the system at run-time to decide how to exploit the parallelism. ARC is a programming model that allows the coexistence of sequential and parallel loads in a cluster environment. Performance studies over ARC using parallel TSP shows the load adaptability of the model. ARC illustrates excellently the problem of load balancing in a heterogeneous cluster environment and its related issues. The ARC model reveals the issues of load balancing on a practical network consisting of several Sun Workstations and IBM RS/6000 machines with widely differing computing capabilities. Students are encouraged to do wide variety of experimentation on a campus network to understand these issues on load balancing and process migration in heterogeneous clusters.

ARC solves the problem of handling dynamically changing loads on the nodes of the network by keeping the program oblivious of these nodes at the time of writing the program. The run-time system shifts the blocks of instructions (Anonymous Remote Instruction blocks) to the lightly loaded nodes on the network. This introduces an interesting problem when these ARC blocks need to communicate with each other the partial results of the computation; which was solved by extending ARC to provide inter-task communication using the concept of Distributed Pipes (DP). The resultant new model was named Anonymous Remote Computation and Communication (ARCC). Iterative grid computation problems, such as the steady state problem from fluid dynamics, were solved over DP to illustrate the linear to super linear speed up, obtained by overlapping computation and communication. Both ARC and ARCC use low level network programming for implementation. ARCC displays the importance of both computation and communication in the context of grid computing. ARCC exhibits clearly the importance of memory, network bandwidth, and computing cycles with reference to grid computing. A number of small experiments each establishing the importance of these three parameters for specific application tasks could be undertaken by students who have read this chapter. This will help the students grasp the interplay between parameters in the performance of the applications. The effect of memory on the application's performance is clearly demonstrated in the ARCC model. The ARCC model can be used by researchers in the Computation Fluid Dyanmics(CFD) community for parallel programming CFD applications on workstation clusters.

To raise the abstraction level, the use of middleware for cluster computing was explored. We built P-CORBA, an extension to the Common Object Request Broker Architecture (CORBA), which introduced the notion of concurrency and two key services, namely object migration and load balancing, into CORBA. P-CORBA was our pivotal work in joining the two areas of object orientation and parallel computing over clusters. The chapter helps grasp the manner in which extensions over CORBA can be designed and implemented.

A preliminary grid model named Sneha-Samuham was built to explore distributed problem solving across wide-area clusters. Sneha-Samuham provides task splitting capabilities, and schedules tasks according to the Grid Computing Capacity Factor (GCCF). A neutron shielding application, implemented to evaluate the performance of the model, showed linear speed up over wide-area clusters. The DP and Sneha-Samuham models were combined to solve iterative grid computing problems over the Internet. Fault-tolerance capabilities were incorporated into the model to handle the dynamic nature of the Internet. Sneha-Samuham and it later version of the model called VISHWA are good models for e-science applications on grids.

At the next level, we built Moset, an anonymous remote mobile cluster computing paradigm. Moset enables the integration of mobile devices (mainly laptop kind of devices) into the cluster. It identifies the key issues that help realize mobile cluster computing, namely connectivity, architecture and operating system heterogeneities, timeliness, load fluctuations, and dynamic network conditions. Moset was tested using a distributed image-rendering algorithm over a simulated mobile cluster model. This document also presents our grand vision of building a mobile grid named DOS Grid, as an integration of computation, data, and service grids. The mobile grid enables any device to access or provide the required computing power, information or other services, from or to the grid.

This monograph also includes a CD containing software developed for the models proposed, namely ARC, DP, P-CORBA and Sneha-Samuham. There are a number of interesting student projects that can be given around these softwares. You can send an email to me to get these projects. Some of them are available at the web page *http://dos.iitm.ac.in/GridComputing*

A number of research scholars working in the area of grid computing will find this monograph useful in generating novel research ideas. The book is primarily an aid for researchers in scientific computing, to program their applications using the software given along with the book.

In its entirety, the monograph is useful as an advanced elective at the postgraduate level. It provides study on the systems developed and their associated software, and students can be asked to produce a term paper consisting of their own experimentation on the software.

The monograph is pertinent for undergraduate students as study material for their grid computing course and project. The material can be used as a self-study elective, since the published journal papers have been interwoven as a collection. It is also a supplemental reference book for teachers. The chapters provide extra study material, and student projects can be given around the software. The suggested chapters for a grid computing course are Chapter 3 (ARC), Chapter 5(ARCC) and Chapter 6 (P-CORBA), though other chapters are also pertinent.

Last but not the least, this work is primarily a joint work with a number of research students who worked at the DOS lab. Their names have been included in the appropriate chapters. My wife, Bhavani, and our daughters, Pooja and Sowmya, have always been supportive of my long hours at work: needless to say they are the peaceful and blissful part of my life.

D JANAKIRAM

Acknowledgements

Parts of the material in the chapters have been taken from the following earlier published papers:

1. R. K. Joshi and D. Janakiram, *TUParset : A Language Construct for system independent parallel programming on loosely coupled distributed systems, UT <http://dos.iitm.ac.in/LabPapers/parset.pdf>* Microprocessing and Microprogramming, Euromicro Journal, 41(1995) pp. 245–259.
2. Rushikesh K. Joshi and D. Janakiram, TU*Anonymous Remote Computing: A Paradigm for Parallel Programming on Interconnected Workstations, UTI* EEE Transactions on Software Engineering, Vol. 25, No.1, Jan, 1999, pp. 75–90.
3. Binu K. J. and D. Janakiram, TU*Integrating Task Parallelism in Data Parallel Languages for Parallel Programming on NOWs,* UT Concurrency–Practice & Experience,. 2000; 12: pp. 1291–1315.
4. Binu K. J., Karthikeyan R. and D. Janakiram, *TUDP: A Paradigm for Anonymous Remote Computation and Communication for Cluster ComputingUT <http://dos.iitm.ac.in/LabPapers/DP.pdf>*T, TIEEE Transactions on Parallel and Distributed Systems, Vol. 12, No.10, October 2001, pp. 1–14.
5. D. Janakiram, A. Vijay Srinivas and P. Manjula Rani, *TUA Model for Parallel Programming Over CORBAUT <http://dos.iitm.ac.in/LabPapers/PCORBApaperJPDC.pdf>*T, T Journal of Parallel and Distributed Computing, Vol. 64, No. 11, November 2004, pp. 1256–1269.

6. D. Janakiram, N.V. Palankeswara Rao, A. Vijay Srinivas and M.A. Maluk Mohamed, TU*Sneha-Samuham: A Parallel Computing Model over Grids, UT* in preceedings of the 2005 International Conference on Grid Computing and Application (GCA '05), June 2005, Las Vegas, USA.
7. M. A. Maluk Mohamed, A. Vijay Srinivas and D. Janakiram, TU*Moset: An Anonymous Remote Mobile Cluster Computing ParadigmUTT,* TTo appear in Special Issue on Design and Performance of Networks for Super-, Cluster-, and Grid-Computing to appear in the Journal of Parallel and Distributed Computing (JPDC).
8. K. Krishna, K. Ganeshan and D. Janakiram, TU*Distributed Simulated Annealing Algorithms for Job Shop SchedulingUT*, IEEE Transactions on Systems, Man, and Cybernetics, No.7, Vol. 25, July 1995, pp. 1102–1109.
9. D. Janakiram, T. H. Sreenivas and Ganapathy Subramaniam, *TUParallel Simulated Annealing AlgorithmsUT <http://dos.iitm.ac.in/LabPapers/%20parallelSAJPDC.pdf>*T, TJournal of Parallel and Distributed Computing, Vol.37, No. 2, 1996, pp. 207–212.

Contents

Abbreviations

ARC	Anonymous Remote Computing
NOW	Network of Workstations
DP	Distributed Pipes
CORBA	Common Object Request Broker Architecture
GCCF	Grid Computation Capacity Factor
MPI	Message Passing Interface
MCC	Mobile Cluster Computing
ARMCC	Anonymous Remote Mobile Cluster Computing
SA	Simulated Annealing
JSS	Job Shop Scheduling Problem
TSP	Travelling Salesman Problem
RPC	Remote Procedure Calls
XDR	External Data Representation
SPMD	Single Program, Multiple Data
MPMD	Multiple Program, Multiple Data
RO	Read-Only
RW	Read-Write
WO	Write-Only
RIBs	Remote Instruction Blocks
REV	Remote Evaluation
NFS	Network File System
PVM	Parallel Virtual Machine

COP	Collection of Processes
FDDI	Fiber Distributed Data Interface
MPVM	Menage Parallel Virtual Machine
LAN	Local Area Network
DSM	Distributed Shared Memories
OBS	Object-based Sub-contracting
RFE	Remote Function Evaluation
ARCPs	ARC Primitives
HPF	Horse Power Factor
HPU	Horse Power Utilization
FSM	Finite State Machine
PTT	Program and Task Table
RIT	Recovery Information Table
RLT	Results List Table
TT	Task Table
ET	Event Table
LP	Linear Predictive
HPF	High Performance Fortran
TCP	Transmission Control Protocol
IGM	Iterative Grid Module
GCT	Grid Computation Task
GCP	Grid Computation Problem
UPT	User Process Information Table
UPBWT	User Processes Blocked for Write Table
GCTST	Grid Computation Task Submitted Table
DPT	Distributed Pipes Table
LCT	Local Coordinators Table

GCWT	Grid Computation Work Table
GCTT	Grid Computation Task Table
IGC	Iterative Grid Computation
CPU	Central Processing Unit
IDL	Interface Definition Language
ORB	Object Request Broker
ARTS	Adaptive Runtime System
MPP	Massively Parallel Processor
WAN	Wide Area Network
POA	Portable Object Adaptor
IIOP	Internet Inter-operability Protocol
MNO	Mobile Network Object
DCOM	Distributed Component Object Model
AMI	Asynchronous Method Invocation
IBM	International Business Machines
GCA	Genetic Clustering Algorithm
GA	Genetic Algorithm
MHz	Mega Hertz
GHz	Giga Hertz
CC	Cluster Coordinator
NAT	Network Address Translation
API	Application Programming Interface
FMI	Friend Machines Interface
GUI	Graphical User Interface
FCT	Friend Clusters Table
STT	Sub-task Table
MH	Mobile Host

MSS	Mobile Support Station
UT	Upper Threshold
LT	Lower Threshold
cc	Coordinators
CT	Computed Tomography
SPARC	Scalable Processor Architecture
CPM	Critical Path Method
TMA	Temperature Modifier Algorithm
LEA	Locking Edges Algorithm
MLEA	Modified Locking Edges Algorithm
VAL	(Lar Vaanhoren, Aarts and Lenstra)
DiPs	Distributed Problem Solver
SSA	Sequential Simulated Annealing
DSO	Distributed Shared Object
CH	Cluster Head
SO	Surrogate Object
PRAM	Pipelined Random Access Memory
GIS	Grid Information Services
MA	Mobile Agent
CERN	European Organization for Nuclear Research
LHC	Large Hadron Collider
LCG	LHC Computing Grid
IPG	Information Power Grid
PPDG	Particle Physics Data Grid
ARCC	Anonymous Remote Computing and Communication
DPS	Distributed Processing System
IPC	Inter Process Communication

RAID	Redundant Array of Independent Disks
CSP	Communicating Sequential Processes
LISP	LIST Processing
NCL	Network Command Language
www	World Wide Web
SCV	Stop Consonant Vowel
lc	local coordinator
sc	system coordinator
GMM	Gaussian Mixture Model
CSM	Constraint Satisfaction Model
NOP	No Parallelism
TD	Task and Data Parallelism exploited
no	Network Overhead
Rn	Resilience to Network Load
l_f	Load Fluctuation Factor
OO	Object Orientation/Oriented
oid	object identity
DII	Dynamic Invocation Interface
XML	Xtensible Markup Language
WP	Worker Process
IIT	Indian Institute of Technology
DNS	Domain Name Server
DTSS	Distributed Task Sharing System
ABZ	Adams, Balas and Zawack method
MSS	Matsuo, Suh and Sullivan method
CA	Clustering Algorithm

D Janakiram is currently Professor at the Department of Computer Science and Engineering, Indian Institute of Technology (IIT), Madras, where he heads and coordinates the research activities of the Distributed and Object Systems Lab. He obtained his Ph.D degree from IIT, Delhi.

He is Founder of the Forum for Promotion of Object Technology, which conducts the National Conference on Object Oriented Technology (NCOOT) and Software Design and Architecture (SoDA) workshop annually. He is also principal investigator for a number of projects including the grid computing project at the Department of Science and Technology, Linux redesign project at the Department of Information Technology, and Middleware Design for Wireless Sensor Networks at the Honeywell Research Labs. He is Program Chair for the 8th International Conference on Management of Data (COMAD). He has guided 5 Ph.D, 23 M.S (research), 50 M.Tech and 31 B.Tech students. He is currently guiding 9 Ph.D and 11 M.S (research) students. He has published over 30 international journal papers, 60 international conference papers and has edited 5 books. He has given several invited and keynote talks at national and international conferences on grid computing.

Professor Janakiram has taught courses on distributed systems, software engineering, object-oriented software development, operating systems, and programming languages at graduate and undergraduate levels at IIT, Madras. He is a consulting engineer in the area of software architecture and design for various organizations. His research interests include distributed and grid computing, object technology, software engineering, distributed mobile systems and wireless sensor networks, and distributed and object databases. He is a member of IEEE, the IEEE Computer Society, the ACM, and is a life member of the Computer Society of India.

Chapter 1

Introduction: Cluster to Grid Computing

1.1 Cluster Computing Models

Parallel computing on interconnected workstations is becoming a viable and attractive proposition due to the rapid growth in speeds of interconnection networks and processors. In the case of workstation clusters, a considerable amount of unused computing capacity is always available in the network. However, heterogeneity in architectures and operating systems, load variations in machines, variations in machine availability, and the failure susceptibility of networks and workstations complicate the situation for the programmer. In this context, new programming paradigms that reduce the burden involved in programming for distribution, load adaptability, heterogeneity, and fault tolerance gain importance. We have identified the issues involved in parallel computing on a network of workstations. The Anonymous Remote Computing (ARC) paradigm is proposed to address the issues specific to parallel programming on workstation systems. ARC differs from the conventional communicating process model as it treats a program as one single entity consisting of several loosely coupled remote instruction blocks instead of treating it as a collection of processes. The ARC approach results in transparency in both distribution and heterogeneity. At the same time, it provides fault tolerance and load adaptability to parallel programs on workstations. ARC is developed in a two-tiered architecture consisting of high-level language constructs and low-level ARC primitives. This chapter describes an implementation of the ARC kernel supporting ARC primitives.

ARC is a pure data parallel approach and assumes that there is no inter-task communication. We have explored the transparent programmability of communicating parallel tasks in a Network of Workstations (NOW). Programs which are tied up with specific machines will not be resilient to the changing conditions of a NOW. The Distributed Pipes (DP) model enables location-independent inter-task communication among processes across machines. This approach enables the migration of communicating parallel tasks according to runtime conditions. A transparent programming model for a parallel solution to Iterative Grid Computations (IGC) using DP is also proposed. Programs written by using the model are resilient to the heterogeneity of nodes and changing conditions in the NOW. They are also devoid of any network-related code. The design of runtime support and function library support are presented. An engineering problem, namely, the Steady State Equilibrium Problem, is studied over the model. The performance analysis shows the speed-up due to parallel execution and scaled down memory requirements. We present a case wherein the effect of communication overhead can be nullified to achieve a linear to super-linear speed-up. The analysis discusses the performance resilience of IGCs, and characterizes synchronization delay among sub-tasks, and the effect of network overhead and load fluctuations on performance. The performance saturation characteristics of such applications are also studied.

Both ARC and ARCC (Anonymous Remote Computing and Communication) use low-level network programming for implementation. In order to raise the abstraction level, the use of middleware for cluster computing was explored and we built P-CORBA. Existing models for parallel programming over Common Object Request Broker Architecture (CORBA) do not address issues specific to parallel programming over NOWs. P-CORBA, a model for parallel programming over CORBA addresses these issues. The transmission and distribution of computing power of a NOW are facilitated by P-CORBA. The main contribution of the work is to bring a notion of concurrency into CORBA. The model illustrates a method for balancing the load on a CORBA-based distributed system. It also provides a new idea for achieving object migration in CORBA. We present detailed performance studies from a prototype of the model that has been implemented. A detailed performance comparison of the model is made with a widely used parallel programming tool, namely Message-Passing Interface (MPI). We demonstrate that in spite of its overheads, CORBA can be used for parallel programming over a NOW and significant speed-ups can be obtained.

1.2 Grid Models

The Sneha–Samuham grid computing model is an attempt to provide an adaptive parallel computing support over computational grids for solving computation-intensive applications. Unlike other grid computing models, Sneha–Samuham provides task-splitting capabilities, wherein the given task is split according to the computational capabilities of the nodes participating in the computation. Aggregating resources in Sneha–Samuham is as simple as making friends by using an instant messenger. The runtime environment of Sneha–Samuham executes a task efficiently by sharing the task among the participating machines, depending on their computation capability, which is measured by using a Grid Computation Capacity Factor (GCCF). The Sneha–Samuham grid computing model has been implemented over a nationwide grid. The model has been evaluated by using neutron shielding simulation application. The results show that it achieves almost linear speed-up. A comparison with MPI shows that Sneha–Samuham outperforms MPI, especially when machines with varying GCCFs comprise the grid. Currently, scientific applications that are purely data parallel and coarse-grained can benefit from Sneha–Samuham.

1.3 Mobile Grid Models

Advances of technology in terms of cellular communications and the increasing computing power of the mobile systems, have made it convenient for people to use mobile systems more than static systems. This has seen the greater use of mobile devices in personal and distributed computing, thus making the computing power ubiquitous. The combination of wireless communication and cluster computing in many applications has led to the integration of these two technologies to emerge as a Mobile Cluster Computing (MCC) paradigm. This has made parallel computing feasible on mobile clusters, by making use of the idle processing power of the static and mobile nodes that form the cluster. In order to realize such a system for parallel computing, various issues such as connectivity, architecture and operating system heterogeneities, timeliness issues, load fluctuations in machines, machine availability variations, and failures in workstations and network connectivities need to be handled. Moset, an Anonymous Remote Mobile Cluster Computing (ARMCC) paradigm is being proposed to handle these issues. Moset provides transparency to the mobility of nodes, distribution of computing resources, and to heterogeneity of wired and wireless networks. The model has been verified and validated by implementing a distributed image rendering algorithm over a simulated mobile cluster model.

Advancement in technology has enabled mobile devices to become information and service providers by complementing or replacing static hosts. Such mobile resources are highly essential for on-field applications that require advanced collaboration and computing. This creates a need for the merging of mobile and grid technologies, leading to a mobile grid paradigm. The key idea in building the mobile grid is to integrate the computational, data and service grids. Thus a mobile device from anywhere and at any time, can harness computing power, and the required resources and services seamlessly. Simultaneously, the device could also be providing location-sensitive data to the grid. We have designed and prototyped a middleware for a mobile grid that transparently manages and bridges the requirement of the mobile users and the actual providers.

1.4 Applications

Simulated Annealing (SA) has been considered a good tool for complex non-linear optimization problems. The technique has been widely applied to a variety of problems. However, a major disadvantage of the technique is that it is extremely slow and hence unsuitable for complex optimization problems such as scheduling. There are many attempts to develop parallel versions of the algorithm. Many of these algorithms are problem-dependent in nature. We present two general algorithms for SA. The algorithms have been applied to the Job Shop Scheduling Problem (JSS) and the Travelling Salesman Problem (TSP), and it has been observed that it is possible to achieve super-linear speed-ups using the algorithm.

Job Shop Scheduling (JSS) belongs to the class of NP-hard problems. There are a number of algorithms in the literature for finding near optimal solution for the JSS problem. Many of these algorithms exploit problem specific information and hence, are less general. However, simulated annealing algorithm for JSS is general and produces better results when compared to other algorithms. But one of the main drawbacks is that the execution time is high. This makes the algorithm inapplicable to large scale problems. One possible approach is to develop distributed algorithms for JSS using simulated annealing. Three different algorithms have been developed, namely Temperature Modifier, Locking Edges and Modified Locking Edges algorithms.

1.5 DOS Grid: Vision of Mobile Grids

We finally present our grand vision of building a mobile grid as an integration of computation, data and service grids. The mobile grid enables any device to access or provide the required computing power, information or other services from or to the grid. The data considered also includes lower-level data collected or aggregated from the sensor devices. The mobile grid requires monitoring data for a variety of tasks such as fault detection, performance analysis, performance tuning, performance prediction, and scheduling. The requirements and essential services are outlined that must be provided by a mobile grid monitoring system. We also present its realization as a peer-to-peer overlay over a distributed shared object space. The proposed mobile grid model visualizes the architecture as a distributed shared object space, wherein all the participating mobile devices are modelled as surrogate objects which reside on the wired network. We illustrate the mobile grid through a mobile health care application.

Chapter 2

Parset: System-independent Parallel Programming on Distributed Systems*

2.1 Motivation and Introduction

During the last decade, a significant amount of interest has been shown in the development of parallel programs on loosely coupled distributed systems. An example of such a system is a set of powerful multi-programmed workstations connected through a local area network. Several mechanisms for performing inter-process communication, synchronization, mutual exclusion and remote accession have been proposed to cope up with the challenges arising out of the distributed nature of these systems. Some examples of these mechanisms are client-server communication, message-passing multiple programs, and Remote Procedure Calls (RPC). An excellent review of such programming paradigms for distributed systems appeared in [1]. Writing parallel programs on loosely coupled systems demands the effective use of these paradigms in the program. For example, in [2], Bal, *et al.* demonstrate how RPCs [6] can be used for writing parallel programs. In addition to making an effective use of these paradigms, several tasks such as creating remote processes on various nodes and writing External Data Representation (XDR) routines have to be performed by the programmer. Distributed programs written with

*Rushikesh K. Joshi, D. Janakiram

these mechanisms are generally difficult to understand and to debug. A programmer can start his processes on heavily loaded nodes, thereby causing severe load imbalances, resulting in under-utilization of the network. This can also adversely affect the performance of other programs running in the network. A key property of distributed systems is that they are open-ended. Various system parameters like node configuration and node availability keep changing over a period of time. In such cases, the programs need to adapt themselves dynamically to the changing system configurations.

Programming tools and languages such as ConcurrentC [5], P4 [3] and PVM [8] exist for developing such parallel programs. These systems have greatly unified the concept of coarse grain parallel programming on several architectures by providing the architecture-independent interface to the programming paradigms in distributed systems. But the programmer has not been relieved of the burden of distributed programming and he needs to write the program as a collection of processes which communicate and synchronize explicitly.

Thus, there is no clear separation between the programmer's concerns and the system's concerns in the present approaches to distributed programming. It has become necessary to provide high-level language constructs which can do this task. These constructs should be provided with adequate low-level runtime support which can achieve the separation between the system's concerns and the programmer's concerns. These language constructs can be provided as extensions to existing programming languages. The programmer can specify his coarse parallel blocks within his program by making use of these high-level language constructs. These language constructs are suitably translated and handled by the low-level system mechanisms. The various advantages of using such constructs are listed below.

2.1.1 Advantages from the System's Point of View

If the selection of nodes for performing computation is made at the programming level, the programmer can write programs which can generate heavy load imbalances in the system. For example, P4 gives the choice of node selection to the user. At the time of node selection, the user may not be in a position to predict the actual load on the selected nodes at the time of execution.

Also, the programs may not make use of dynamically changing loads on the machines due to its rigid process configuration. For example, a program in PVM may create a fixed number of processes. In such a case, the program will not be able to utilize the additional capacity in the system if some nodes become lightly loaded when the program actually starts executing.

With appropriate high-level language constructs, the programmer can only express the willingness of parallel execution. With this, the programs need not be modelled as a pre-configured collection of processes. This provides maximum flexibility to the system to make effective use of the available resources.

2.1.2 Advantages from the Programmer's Point of View

(i) The user can be relieved of the burden of creating processes and performing explicit communication and synchronization among them.

(ii) The number of available nodes and their interconnection pattern vary from one distributed system to the other, and also from time to time in a single distributed system. The load on the machines frequently keeps changing. In such a case, if a program is written as a collection of a fixed number of processes, it cannot make use of the dynamically changing loads in the system. By using the high-level language constructs, the programs can be written in a system-independent fashion, thereby making them scalable and portable.

(iii) Several programming errors, which occur during programming inter-process communication, synchronization, termination etc., can be avoided by programming with such language constructs.

In this chapter, we present a language construct called *parset* and the low-level runtime support for implementing it. Parsets can be used for expressing coarse grain parallelism on distributed systems. The parset construct consists of a data structure and a set of functions which operate on this data structure. The construct has been specially designed for capturing several kinds of coarse grain parallelism occurring in distributed systems. The use of parsets relieves the programmer of the burden of handling the remote processes, inter-process communication, remote procedure calls, etc. A low-level distributed parset kernel creates sub-tasks, locates suitable remote nodes, and gets the code executed on the remote nodes. This makes the programs that are written using parsets, scalable over varying system parameters. Thus, parsets draw a clear distinction between the system's concerns and the programmer's concerns.

We develop the parset constructs in Section 2. Parsets can capture both SPMD (Single Program, Multiple Data) and MPMD (Multiple Program, Multiple Data) kinds of parallelism. This is explained in Section 3. We implemented the parsets on a network of workstations. This implementation, with a case study of an application program, is discussed in Section 4.

2.2 Semantics of the Parset Construct

In this section, we first describe the parset data structure with the basic operations which manipulate the data structure. When functions receive parsets as their arguments, they derive special meanings. The function semantics on parsets are explained subsequently. Finally in this section, a special case of parset called *indexparset* is described.

2.2.1 The Parset Data Structure

Parset is a set type of data structure. The elements of a parset can be:

- Basic data types,
- Untyped,
- Functions.

When the elements of a parset belong to a basic data type, it is called a *simple parset*. When the elements are untyped, it is an *untyped parset*. A *function parset* has functions as its elements. Simple parsets can be used to express the SPMD kind of parallelism. This is discussed in detail in Section 2.3.1. Untyped parsets find applications in expressing MPMD kind of parallelism using polymorphic functions [4]. Function parsets are the most general ones, and by using them, it is possible to express the MPMD parallelism in a general way. The MPMD parallelism using parsets is discussed in detail in Section 2.3.2.

A parset is kept logically ordered on the basis of the entry of its elements on a first-come-first-served basis. Cardinality is an attribute associated with a parset. The cardinality of an empty parset is zero. A typed parset declares the type of the elements held by the parset. For example,

parset P of int;

declares a parset P of elements of type integer.

The operations that can be performed on parsets are **insert(), flush(), get(), delete(), getcard()** and **setcard().** The functions that can be executed on parsets have their arguments tagged as **RO** (read-only), **RW** (read–write) or **WO** (write-only). This scheme is very similar to the Ada language approach which places the reserved keywords IN, *OUT,* and *INOUT* before the arguments. The semantics of these tags are as follows:

WO : The argument is only modified inside the function but not read.

RO : The argument is passed to the function only for reading.

RW : The argument is read as well as written to inside the function.

When functions taking parsets as arguments are executed, the arguments are locked in a proper mode. For example, if an argument is tagged as **RO,** it is locked in read mode. This tagging scheme allows the exploitation of parallelism in control flow. When two functions are sequenced one after the other in a program, they can be run in parallel if the earlier function does not lock the arguments needed for execution of the second function. For example, if two functions take the same argument tagged as **RO**, they can be executed in parallel. On the other hand, if the argument is tagged as **RW**, unless one function releases the locks on the argument, the other function cannot start execution. The argument tagging performs an important role in the case of parsets and has implications in implementation. This is discussed in detail in Section 2.4.4.2.

The operations which manipulate the parsets are as follows:

insert (**WO** P, **RO** i, **WO** order)

Inserts an element *i* in the parset ***P*** as its last element. The cardinality of ***P*** increases by 1 after the insertion operation. The argument order returns the order of the inserted element. ***P*** is tagged as **WO** here as the function writes an element into the parset but does not read any element from it.

flush (**WO** P)

Flushes the parset ***P***. After flushing, the cardinality of ***P*** becomes zero.

delete (**WO** P, **RO** n)

Deletes the nth element from the parset ***P***. The delete operation leaves the order of the parset intact.

get (**RO** P, **RO** n, **WO** element)

The argument ***element*** returns the nth element of the parset ***P***.

getcard (**RO** P, **WO** c)

The argument c returns the cardinality of the parset ***P***.

The above operations provide the means for manipulating the elements of a parset. When functions are called on parsets, they acquire special meanings. This is explained below.

2.2.2 Function Semantics on Parsets

When a parset is passed as an argument to a function, three possibilities exist for the execution of the function. The function can execute in parallel on each element of the parset. This is the first type of execution. The function can also execute sequentially on each element of the parset one after the other. This is the second type of execution. In the third type of execution, the function takes the parset as a simple argument for further processing within the function. In order to differentiate between these three function types, the two keywords **par** and **seq** are used. We now illustrate the three types of function calls with simple examples.

par function call:

Example: par process (**RO** P);

The function process() is applied to each element of ***P*** in parallel. Each activation proceeds asynchronously. If a function has more than one parset as its arguments, then the cardinalities of all of them must be the same. This will enable a particular function activation to pick up the corresponding element from each parset. The ***par*** function call exploits the data parallelism expressed in a parset.

seq function call:

Example: **seq** print (**RO** P);

The function print() is applied to each element of ***P*** sequentially in the order of the elements.

Ordinary function call:

Example: myprint (**RO** P);

Here no keyword is prefixed to the function call. Hence this is treated as an ordinary function call, and the parset ***P*** is passed just as a plain argument to it.

As an example, the following can be the description of myprinto:

```
Function myprint  (RO  P)  {
  seq print  (P);
}
```

A special function called 'myid' is provided to identify the element of the parset on which the present function activation is operating. This function returns the order of the element of the parset on which a *par* or a *seq* function call is operating.

2.2.2.1 *Defining the Functions which Execute on a Parset*

When a *par* or a *seq* function is called with a parset as its argument, each activation of that function receives one element of the parset. Hence, the function is defined for one element of the parset. On the other hand, a function, which takes a parset as a plain argument as in the case of an ordinary function call, declares its argument type the same as the parset type itself. The following example illustrates the difference. In this example, add() concatenates strings and howMany tells the cardinality of a collection of strings.

```
Function MassConcat()  {
    .  .  .
    parset  P  of  string  =  „Work",  „Think",
    „Speak";
    parset  Q  of  string  =  „hard",  „deep",
    „truth";
    parset  R  of  string;
    int  n;
    .  .  .
    .  .  .
    .  .  .
    par  add  (P,  Q,  R);
    howMany  (R,  n);
    .  .  .
  }
```

```
Function add (string RO x, string RO y,
string WO z)
  {
  concat (x, y, z);
  }
Function howMany (parset RO WO n)
  {
  getcard (strset, n);
  }
```

Each activation of function add() binds an element of parset ***P*** to argument x, and the corresponding element of parset ***Q*** to y. The returned argument z is bound to the corresponding element of *R*. The cardinality of the parset ***P*** and ***Q*** must be the same in this case.

2.2.2.2 *Concurrent Execution of Multiple Function Calls on Parsets*

There can be situations wherein a parset, which is an output parameter of one function call, becomes an input parameter in a subsequent parset function call. This offers additional possibilities of concurrent execution. This is explained in the following example:

```
Function Encourage () {
  . . .
  parset E of employee, A of assessment, R
 of reward;
    par assess (RO E, WO A);
    par encourage (RO A, WO R);
    . . .
}
```

In the function Encourage () as given above, the employee records are assessed so as to encourage the employees by offering them suitable rewards. The output of assess (), which is parset A, is an input to Encourage (). As soon as the function assess () finishes with any one of its multiple activations, the **WO** lock on the corresponding element of the parset is released. After the release of the lock, the next function Encourage () acquires the **RO** lock for that particular element of the parset. Once the lock is acquired, the function can start its execution. Thus, multiple functions can execute concurrently providing additional parallelism in control flow.

2.2.3 Indexparset

The indexparset is a special case of parset. It holds no elements but carries an index. The index can be seen as cardinality of the indexparset. A function call on an indexparset is activated index number times. Hence the indexparsets can be used when multiple activations of the same function are required. An indexparset can be declared as:

indexparset *I*;

Only three operations, namely **setcard(), getcard(),** and **flush** (), are performed on an indexparset. The last two are the same as described in Section 2.2.1 on parsets. Operation **setcard** sets the cardinality of the indexparset *I* to a given value *c* and is defined as:

setcard (**WO** *I*, **RO** *c*);

When an indexparset becomes an argument to a ***par*** or a ***seq*** function call, each function activation receives an integer which represents the order of that particular activation. The following example demonstrates the use of an indexparset. It collects the status of distributed resources in a parset called StatusSet.

```
Function CollectStatus {
  indexparset I;
  parset StatusSet of int;
    setcard (I, 10);
    par myread (I, StatusSet);
      / * The activations of myread collect
      the status of 10
```

```
            resources */
      …..
      }
    Function myread (int RO ResourceId, int
    WO s)
    {
      / * ResourceId = current function activation
      number * /
      ReadStatus (ResourceId, status);
      S = Status;
    }
```

2.3 Expressing Parallelism through Parsets

Parsets can be employed for expressing both SPMD and MPMD kinds of parallelism. This is discussed in the following sections.

2.3.1 Expressing the SPMD Parallelism through Simple Parsets

The SPMD parallelism can be expressed by using simple parsets with ***par*** function calls. When a ***par*** function is called on a simple parset, the function executes in parallel on different elements of the parset. The simple parsets can be created in two ways. An empty parset can be declared initially and the elements can be inserted into the parset explicitly by insert calls. The other method of creating simple parsets is to convert an array of elements into a parset with a grain control mechanism. These two methods of creating simple parsets are explained in Sections 2.3.1.1 and 2.3.1.2, respectively.

2.3.1.1 Creation of Simple Parsets using Explicit Insert () Calls

First a simple parset of the desired data type is declared. Multiple data belonging to the same data type can now be added to the parset using the *insert ()* function. Then a function can be executed on this parset by a ***par*** function call. In order to express the SPMD parallelism in the processing of arrays, one may create a parset and explicitly insert the array elements into the parset with this method. An easy way to convert an array into a parset is through the grain control mechanism, which is specially designed for this purpose.

2.3.1.2 Conversion of Arrays into Simple Parsets by the Grain Control Mechanism

Through this mechanism, one can indicate the granularity that is desired to build such a parset out of an array. The mechanism consists of two constructs, namely *granularity,* which is a metatype, and a function *CrackArray(). The* granularity works as a metatype in the sense that its value is a data type.

As an example, we may specify a granularity of *int[100]* to convert an array of type *int[1000]* into a parset. The parset will have ten elements, each of type *int[100]* as specified by *granularity*. It is possible to covert a multi-dimensional array into a parset by cracking the array in any dimension. For example, we may specify a granularity of *int[25][100]* to convert a row major array of type *int[100][100]* into a parset. In this case, the parset will have four elements, each of type *int[25][100].*

We take the following image transformation example to demonstrate the use of this mechanism. In this example, the array named *A* is converted into a parset. The array *A* consists of 1000 elements. Each element of the parset is constructed by combining 100 elements of the array. Thus the parset will have ten elements in it.

```
Function ProcessArray () {
  int A [1000];
  granularity g = int [100];
  parset P, Q of g;
    CrackArray (A, P, g);
    par transform (P, Q);
```

```
        par plot (Q);
        flush (P);
    }
```

Converting a data structure like an array into a group of several grains of specific granularity becomes possible with the metatype 'granularity'. The target parset is declared as a simple parset of the same data type as that of the granularity. The array is converted into a parset by using the function *CrackArray ()*. The function takes three arguments: the source array, the target parset handle and the granularity. When the function returns, the parset handle corresponds to the new parset that is built out of this array.

After the conversion, there are two ways to access the array. One is by manipulating the new parset handle, and the other is by directly using the array name. The array name refers to the copy of the array which is local. The parset handle refers to the copy of the array which may be scattered in the network. Now if both the handles are allowed to manipulate the array, it will create inconsistencies between these two copies. Hence till the parset handle is active, the array must be accessed only through the parset handle. However, a *flush ()* call on the parset handle has the special function of deactivating the handle and storing back the new values of the array into its local copy. After the flush, the array can be referenced in the normal way.

By setting the granularity, the user can set an upper bound on the number of processors that can be utilized by the underlying parset kernel for an execution. For example, for a single-dimensional array of size 1000, the granularity may be set to 100 or 250, thereby creating parsets of size 10 or 4. Thus a varying degree of parallel execution on a parset can be obtained by using granularity.

The grain control mechanism thus achieves two goals. First, it allows an array to be treated directly as a parset without the need for multiple *insert ()* calls. Secondly, it can control the degree of parallel execution of a function on an array.

Dynamic Specification of Granularity

When a granularity variable is declared as an array, the dimensions of the array can be specified during runtime. The following example demonstrates this:

```
Function ProcessArray () {
  int A [1000];
  granularity g = int [];
  parset P, Q of g;
  int dim;
    dim = 100;
    CrackArray (A, P, g [dim]);
    par transform (P, Q);
    par plot (Q);
    flush (P);
}
```

It can be noted that the value of the granularity variable has been declared as an integer array of single dimension without mentioning its dimension value. The dimension of 100 has been specified during runtime in this example.

2.3.2 Expressing MPMD Parallelism through Parsets

The MPMD parallelism can be expressed with parsets by using two mechanisms. The first is to use polymorphic functions with untyped parsets. It offers a limited way of expressing MPMD parallelism, and can be readily implemented in a language which provides function polymorphism. The other mechanism is a general one and uses function parsets. These mechanisms are described in the following sub-sections.

2.3.2.1 MPMD Parallelism through Polymorphic Functions

Untyped parsets can be used for expressing MPMD parallelism. In a typed parset, only the elements of the specified type can be inserted, whereas in untyped parsets, elements of any type can be inserted. An untyped parset declaration does not mention the type of its elements. As an example, an untyped parset Q can be declared as:

parset Q;

In this case, a parset is created as a collection of data belonging to different types. A polymorphic function identifies the type of each element and executes the required code on it. For example, with a function call such as **par** process (Q), various activations of the function process() may receive arguments of different types. In this way, we can obtain the concurrent execution of different codes on different data.

But this mechanism cannot fully capture the MPMD parallelism for the following reason. Always the same piece of code is executed on two different elements of a parset if they are of the same type. So we cannot execute different codes on the elements of the same type by this mechanism. By using the function parsets, this limitation can be overcome.

2.3.2.2 MPMD Parallelism with Function Parsets

A function parset can hold a collection of functions. Another parset is used to hold the corresponding argument set. The cardinalities of these two parsets must be the same. The function parset can now be invoked on the corresponding argument parset to achieve the general MPMD parallelism. The following example shows the structure of such a program:

```
Function MPMD-Prog-Structure () {
  int d11, d12, d3;
  char d2;
  parset F of func = {f1 (int, int), f2
 (char), f3 (int)};
  parset Dl = {dll, d2, d3};
  parset D2 = {d12, NIL, NIL};
    par (F) (Dl, D2);
}
```

```
Function f1 (int RO d11, int RO d12) {
. . .
}
Function f2 (char RO d2) {
. . .
}
Function f3 (int RO d3) {
. . .
}
```

In the above example, function f1() will take its first argument from the first element of parset *D1,* and the second argument from the first element of parset *D2.* Similarly, the functions f2() and f3() pick up their arguments from *D1* and *D2.*

2.4 Implementing Parsets on a Loosely Coupled Distributed System

We discuss an implementation of the parset constructs on a network of workstations consisting of Sun 3/50 and Sun 3/6Os, running SunOS version 4.0.3. The environment supports the Network File System (NFS). TCP/IP has been used for inter-process communication. The parset constructs are provided as extensions to the C language. A parset pre-processor translates the parset constructs into C code. An overview of the implementation is now given followed by the description of various components. We studied the performance of an application using the implementation.

2.4.1 Overview of the Implementation

The heart of the implementation consists in a distributed parset kernel. The kernel is divided into resident and volatile parts. Each parset has an associated process called *P-Process* to maintain and manipulate the elements of the parset. A ***par*** function call on a parset is executed with the help of separate processes called

E-Processes. *P-Processes* and *E-Processes* form the volatile kernel. *P-Processes* reside on the same node where the user program resides. *E-Processes* reside on different nodes to exploit the parallelism.

The resident kernel consists of daemon processes which are started during the boot-up time on the machines that are willing to participate in the execution of programs that use parsets. The copy of the resident kernel is obtained by nodes from a designated node using the NFS. In this way, both diskless as well as diskful machines can obtain the resident kernel. The resident kernel manages the *P-Processes* and the *E-Processes*. It provides an interface to user programs to create the parset processes and to execute various functions on it.

2.4.2 The Parset Preprocessor

Figure 2.1 shows the functionality of the parset preprocessor. The high-level user program is provided with clean parset constructs, as described in the earlier sections, which hide the underlying distributed implementation. The parset preprocessor then translates the user program into a low-level C code, which makes calls to the resident kernel and the parset processes.

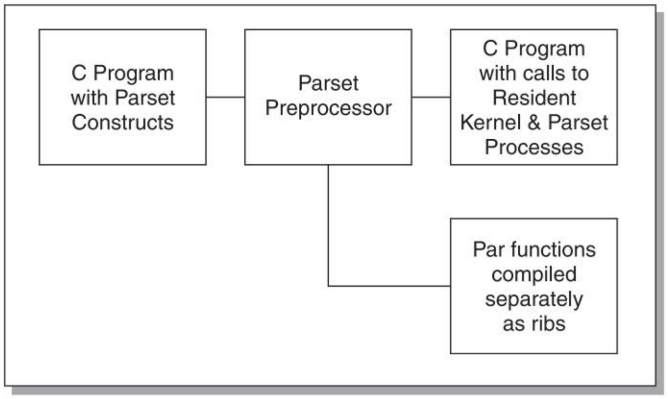

C Program with Parset Constructs
Parset Preprocessor
C Program with calls to Resident Kernel & Parset Processes
Par functions compiled separately as ribs

FIG. 2.1

The Parset Preprocessor

The parset preprocessor identifies the *par* function calls separately as Remote Instruction Blocks (RIBS). An RIB contains a code which can be migrated at runtime to a remote node. The migration of the RIB code is the migration of the passive RIBs and not active processes or tasks. The RIB is given control for actual execution at the remote node. RIBs are named and compiled separately by the preprocessor.

2.4.3 Remote Instruction Blocks (RIB)

The RIB facility was developed during the course of this implementation. The facility is similar to Remote Evaluation (REV) [7] developed by Stamos and Gifford. With the RIB primitives, as opposed to the RPC primitives [6], we can migrate a code to be executed to a remote site at runtime and get it executed there.

The parset preprocessor names and compiles the RIBs separately. Whenever a block has to be executed on a remote node, its name and address is made known to the remote node. The remote node can access the RIBs by using the NFS, and can execute it as and when required.

2.4.4 The Distributed Parset Kernel

The parset kernel is distributed over the network. It consists of a resident and a volatile part. The resident part of the kernel is always present on all the nodes that participate in distributed execution. The volatile part is dynamically created on selective nodes depending upon the requirements. Figure 2.2 shows the organization of various components in the implementation. The functionalities of each are described in the following sections:

2.4.4.1 The Resident Kernel

The resident kernel performs the low-level system-dependent tasks which deal with the primitives for RIBs, creation and termination of *P-Processes* and *E-Processes,* etc. It is responsible for selecting the lightly loaded nodes for execution and managing the distributed

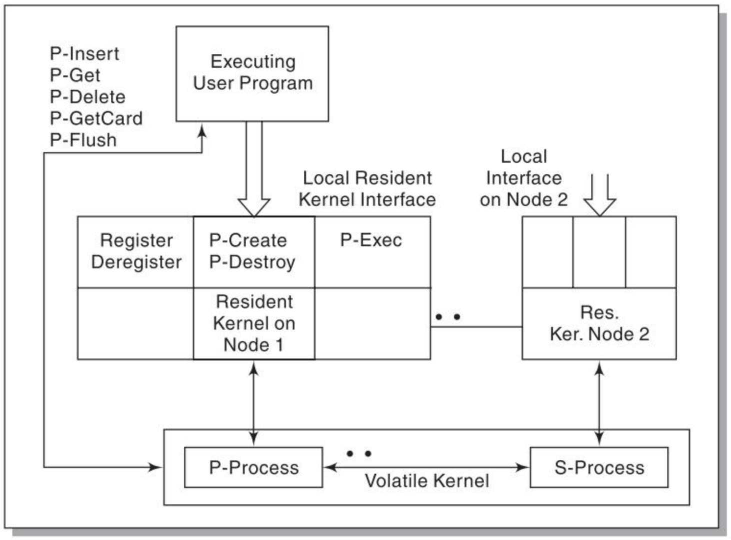


FIG. 2.2

The Parset System Components

execution. Each local resident kernel contacts other resident kernels and finds about the lightly loaded nodes. Achieving the program scalability is also a function of the kernel.

The resident kernel provides an interface to low-level user programs. The interface consists of the calls for registration and de-registration of a user process, creation and destruction of parset processes, and notifications of function execution on parsets. When an executing user program declares a parset variable, the resident kernel spawns a parset process called *P-Process*. This *P-Process* manages the parset on the node where the user program runs. Any further manipulations performed by the user program on this parset with **insert**(), **get**(), **delete**(), **getcar**() and **flush**() calls, are directed to the corresponding *P-Process* bypassing the resident kernel.

During the remote execution of a function in a *par* call, the resident kernel creates new *E-Processes,* or locates free executor processes depending on their availability on the nodes suitable for the execution. For a particular function execution, the resident kernel helps *P-Processes* and the corresponding *E-Processes* to synchronize. After the synchronization, *P-Processes* send the required parset elements to *E-Processes.*

In order to improve the execution efficiency, the kernel can combine several elements in a parset together to form a super-grain. It can be noted that the super-grain formation is different from the grain control mechanism. The latter is a mechanism meant for the user whereas the former is used by the kernel, making it transparent to the user. The resident kernel guides the volatile kernel for the super-grain formation.

2.4.4.2 The Volatile Kernel

The volatile part of the kernel consists of *P-Processes* and *E-Processes,* as mentioned earlier. For a function execution, the *P-Processes* which correspond to parsets involved in the execution, perform the required inter-process communication with the allotted *E-Processes.* In this fashion, all *P-Processes* and *E-Processes* proceed concurrently. An *E-Process* can be allocated for other incoming execution requests after the current request is completely processed. Similarly, a *P-Process* is derailed by the resident kernel when it receives a **destroy** () call.

As discussed earlier, a *P-Process* manages one single parset variable. A parset variable is a collection of multiple grains (elements). When a *par* or a *seq* function takes a parset variable as its argument, it specifies the parset variable as an **RO**, **WO**, or **RW** variable depending on its usage inside the function. At the time of actual execution, the elements of the parset have to be locked according to these specifications in order to exploit the concurrency as discussed in Section 2.2. In case of multiple parallel activations of a function, each activation operates on a different grain. Hence each activation can proceed independently as long as it can obtain the required locks.

At first sight, it appears that in the place of **WO** locks, one may use **RW** locks, thereby eliminating the need for additional **WO** locks. This is based on the fact that along with a write permission, there is no harm in granting a read permission also. But this is not so in reality because when an argument gets a read lock, it has to be moved to the remote site where the function is executing. But a **WO** argument need not be moved to the site of the remote function, since the function does not read the value of this argument. Thus these three distinct locks help in minimizing the communication overhead.

With an execution request of a *par* or a *seq* function, a parset process first establishes contact with the remote *E-Process* through the resident kernel. Then it continues to set the locks on the grains. As and when a lock is set, **S** the following action is taken corresponding to each lock:

RO: 1. Send a copy of the grain to the remote *E-Process*.

2. Release the lock on the grain.

WO: 1. Receive the grain value from the remote *E-Process*.

2. Update the grain value in the parset.

3. Release the lock.

RW: 1. Send a copy of the grain to the remote *E-Process*.

2. Receive the grain value from the remote *E-Process*.

3. Update the grain value in the parset.

4. Release the lock.

2.4.5 A Case Study

In this section, we discuss a simple case study of the image transformation problem encountered in computer graphics. Two examples of such transformations are rotation and dragging. Figure 2.3 shows the test program for image transformation using parsets. The program was run on a network of Sun workstations. Since the transformation problem was an SPMD type of problem, we used the techniques presented in Section 2.3.1.2.

The following observations can be made with respect to the case study program:

- The program only expresses the parallelism present in solving the graphics rotation problem without any explicit use of system-dependent primitives.

```
#define TotalPoints 20000
#define GrainSize 400
typedef struct {int x, y;} point;
typedef point grain [GrainSize];
Main:
  point image [TotalPoints];
  granularity G = grain;
  parset P of G;
    read-image (image);
    CrackArray (image, P, G);
    par transform (P);
    flush (P);
EndMain
Function: transform (grain RW image-
grain) {
  int i;
    for (i = 0; i < GrainSize; i++)
    image-grain = rotate (image-grain);
}
```

FIG. 2.3

The Image Transformation Problem

- The user can control the size of the grain by specifying the grain size in the program.
- The function transform() will be compiled separately as an RIB so that it can be migrated to a remote node for execution. The system will make appropriate choice of nodes and the mechanism is completely transparent to the user program.
- The program is easy to understand and to debug.

- The program can be executed on a distributed system supporting the parset construct and hence can be ported easily to other systems.
- In the case of node failures, the kernel can reassign the sub-tasks which remain unevaluated to other nodes without the involvement of the user.

The above program has been executed on the current implementation of parsets on Sun 3/50s and 3/60s, and the performance has been measured.

The Program Performance

Table 2.1 summarizes the performance of the program with various problem sizes on a varying number of nodes. As the problem size increases, it is possible to use a higher number of nodes. The upper bound on the number of nodes that the system can choose for a given problem size is dictated by the grain size as discussed in Section 2.3.1.2.

Table 2.1 Performance of the Program

Problem size	*Tseq (sec)*	*Grain size*	*#nodes N*	*Tpar (sec)*	*Speed-up*	*Utilization*
2000	17.85	1000	2	10.71	1.66	83
		500	4	7.50	2.38	59
4000	36.02	2000	2	20.36	1.76	88
		1000	4	12.84	2.80	70
10000	90.04	5000	2	48.00	1.87	93
		2500	4	26.48	3.40	85
		1250	8	16.93	5.31	66

From Table 2.1, it can be observed that as the problem size increases, it is possible to employ more nodes thereby achieving better speedup and utilization.

A Comparison with PVM

The same application program has been implemented using PVM. Table 2.2 shows a comparison between parsets and PVM. The execution speeds using parsets compare favourably with PVM. A negligible average overhead of less than two seconds was observed in the implementation of parsets.

Table 2.2 A Comparison with PVM

Problem size	*Grain size*	*#nodes N*	*Timings for parsets (sec)*	*Timings for PVM (sec)*
2000	1000	2	10.71	10.04
	500	4	7.50	5.78
4000	2000	2	20.36	19.02
	1000	4	12.84	10.52
10000	5000	2	48.00	46.28
	2500	4	26.48	24.26
	1250	8	16.93	14.28

The Scalability Test

The image array was converted into a parset with a grain size of 400 points. As the load on different nodes varied, the system automatically chose the nodes for executing the program. During different runs of the program, the system used nodes from 4 to 10 depending on their availability and load. The results are given in Table 2.3. From this table, it can be seen that the parset constructs provide a high degree of scalability to the program without

Table 2.3 Scalability of the Program Problem Size: 20,000 Points, Sequential Time: 179.99 sec

#nodes	*Parallel time (sec)*	*Speed-up*
2	93.94	1.91
4	54.50	3.30
10	25.44	7.07

significant overheads. The scalability is completely transparent to the user. The system is able to handle the node availability dynamically, making use of the nodes as and when they become available on the network.

2.5 Discussion and Future Work

The parsets provide a clear separation between the system-dependent tasks and the expression of parallelism. The programmer need not know about the underlying distributed system model. The user program has to only express the willingness for parallel execution through the parset constructs. The decision of the degree of parallelism that is actually exploited is delayed until runtime. This decision is taken by the distributed parset kernel. For example, in the case of high activity on the network, the same program still runs on a fewer number of nodes, even on a single node. The kernel takes care of all the system-dependent tasks such as node selection, process creation, remote execution and synchronization.

Each element of a parset is a potential source of concurrent execution. A *par* function call captures the data parallelism that exists among the various elements of a parset. On a single element of a parset, two separate functions can execute concurrently if they obtain the necessary lock on that element. In this way, multiple function calls on parsets capture the parallelism present in the control flow.

Fault tolerance is an important issue in distributed systems. The parsets can be made to tolerate node failures in the midst of function execution. This can be ensured at the kernel level with the help of the locking mechanism without the involvement of the high-level program. Whenever a node failure is detected, a new activation for the function executing on that node is spawned on a healthy node. The locks are not released till the activation returns successfully.

The RIBs used in this implementation were designed for a homogeneous system. We are planning to extend them for heterogeneous systems. This will enable us to run the parset programs over a wide area network and exploit the computation power that is available over heterogeneous systems.

2.6 Conclusions

We presented the parset language construct for writing coarse grain parallel programs on distributed systems. Parsets achieve a clear separation between the system's concerns and the programmer's concerns. Programmers can write neat and simple parallel programs on distributed systems by using the parset constructs. A parallel program written with parsets becomes scalable and easily portable to other distributed systems supporting parsets. Language constructs like parsets can make parallel programming on distributed systems an easy task. The parset approach can prove to be an extremely promising direction for parallel programming on distributed systems.

References

1. Andrews, G.R., "Paradigms for Process Interaction in Distributed Programs", *ACM Comput. Surveys,* Vol. 23, No. 1, pp. 49–90, March 1991.
2. Bal, H.E., R.V. Renesse and A.S. Tanenbaum, "Implementing Distributed Algorithms Using RPC", *Proceedings of AFIPS National Computer Conference,* Chicago, Illinois, June 15–18, AFIPS Press, Reston, Virginia, Vol. 56 pp. 499–506, June 15–18, 1987.
3. Butler, R. and E. Lusk, "User's Guide to the P4 Parallel Programming System'', Technical Report ANL92/17, Argonne National Laboratory, October 1992.

4. Cardelli, L. and P. Wegner, "On Understanding Types, Data Aabstractions and Polymorphism", *ACM Comput. Surveys,* Vol. 17, No. 4, pp. 471–522, December 1985.

5. Cmelik, R.F., N.H. Gehani and W.D. Roome, "Experience with Multiple Processor Versions of ConcurrentC", IEEE *Trans. Soft. Eng.,* Vol. 15, No. 3, pp. 335–344, March 1989.

6. Nelson, B.J., "Remote Procedure Calls", *ACM Trans. Comput. Sys.,* Vol. 2, No. 1, pp. 39–59, February 1984.

7. Stamos, J.W. and D.K. Gifford, "Remote Evaluation", *ACM Trans. Prog. Lang. Syst.,* Vol. 12, No. 4, pp. 537–565, October 1990.

8. Sunderam, V.S., "PVM: A Framework for Parallel Distributed Computing, Concurrency", *Practice and Experience,* Vol. 2(4), pp. 315–339, December 1990.

Chapter 3

Anonymous Remote Computing Model*

3.1 Introduction

Workstation clusters are becoming increasingly popular in academic and scientific computing. The processing power of the workstation has witnessed tremendous growth, resulting in clusters of workstations possessing the computing power of a supercomputer. Many approaches have been proposed for improving the utility of workstation clusters in recent times, with the notable ones being the Berkeley NOW project [3] and the Batrun project at Iowa [41]. Broadly, these approaches can be classified into three levels–the operating system level, the resource level, and the programming language level. Some examples of operating system level approaches are the Amoeba distributed operating system [30], the V Distributed System [15], the Coda distributed file system [33], Sun's NFS support, the load leveller on IBM RS/6000 system, and the Batrun's Distributed Processing System (DPS). Cooperative file caching [16] and Reduntant Array of Independent Disks (RAID) systems [3] are examples of the resource level approach. At the programming language level, many programming languages and paradigms have been proposed for loosely coupled distributed systems from the point of view of writing distributed programs as well as developing client-server applications [4], [6].

*Rushikesh K. Joshi, D. Janakiram

Parallel programming on workstation systems is a relatively new field and has been an attractive proposition ever since it was used in some of its early forms such as the RPC thread-based system of Bal, Renesse, and Tanenbaum [8]. Other examples of parallel programming approaches on workstation systems are the Inter Process Communication (IPC) [38] based message-passing systems, RPC thread-based systems [32], distributed shared memories [39] such as Mether system [28], programming tools such as PVM [40] and P4 [11] and programming languages such as Orca [7] and Linda [12].

Most of the existing parallel programming approaches on workstation systems follow the Collection of Processes (COP) model in which processes communicate via message abstractions or shared memory abstractions. Although this model has been widely used in the context of tightly-coupled systems such as multiprocessors, it has several drawbacks when applied to loosely coupled open-ended workstations. Each workstation typically runs its own operating system and provides a multi-user environment. Workstations in a network can be heterogeneous. Loads on workstations keep changing from time to time. Moreover, it may not be possible to support a common file system for all workstations in a domain. Network links and workstations themselves are prone to failures. The number of workstations in a domain keeps changing. These factors pose serious difficulties from the point of view of open-endedness, fault tolerance, load adaptability and programming ease, when the COP model is adopted. For example, as the load on the workstations keeps changing dynamically, it is not possible for the programmer to know which of the nodes will have lighter load when the program starts executing (since the load varies dynamically) and start the processes on these nodes. Dynamic load variations may slowdown the performance of a process that has been mapped onto a particular workstation. Similarly, when a node fails during the execution of a program, a process' state is lost, which requires re-execution of the entire program. Workstations may get connected to a domain during the execution of a program which may not be able to utilize the additional computing power. Such situations make parallel programming on workstation clusters a complicated task. In order to reduce the burden of programming on these systems, new paradigms are required.

In this chapter, we present the ARC approach to parallel programming on workstation systems that attempts to address these issues at the language level. The ARC computational model takes a diversion from the conventional COP model. A program is treated as one single entity made up of multiple loosely coupled RIBs, which are open program fragments that are heterogeneous and fault-tolerant. ARC makes distribution and heterogeneity transparent to the programmer. An ARC task, namely RIB, may be sent by the ARC system for remote execution, or executed on the host node itself if a suitable remote node is not available. An RIB is open in the sense that it may migrate for execution to an anonymous workstation of a different architecture.

ARC provides a two-tiered architecture. At the lower layer, the primitives which are built over a kernel provide the basic support for anonymous remote computing while at the upper layer, various easy-to-use high-level programming language constructs are supported. The ARC kernel provides support for migration, fault tolerance, heterogeneity, load adaptability, and asynchronous intimations for availability, thereby keeping the programming layers very elegant. The ARC model has been implemented on a network of workstations consisting of Sun 3s, Sun Sparcs, and IBM RS/6000 workstations. The network has two independent file systems. The RS/6000s are connected via a 100 MB/sec FDDI (Fiber Distributed Data Interface) network and the Suns are connected via a 10 MB/sec Ethernet.

The rest of the chapter is organized as follows. Section 3.2 discusses the issues involved in parallel programming on workstation clusters followed by an overview and analysis of the existing approaches in Section 3.3. The computational model of ARC is presented in Section 3.4. Section 3.5 discusses the two-tiered ARC architecture. In Section 3.6, the implementation of ARC is presented, followed by the performance study in Section 3.7.

3.2 Issues in Parallel Computing on Interconnected Workstations

Parallel computing on tightly-coupled distributed systems has so far been widely popular. However, recent advances in communication technology and processor technology make parallel programming on loosely coupled distributed systems, especially on interconnected workstations, a viable and attractive proposition. Several key issues distinguish parallel computing on workstation clusters from that of tightly-coupled massively parallel architectures. We discuss these issues in detail in the following four major categories:

1. Changing loads on the nodes of the network,
2. Changing node availability on the network,
3. Differences in processor speeds and network speeds,
4. Heterogeneity in architecture and operating systems.

3.2.1 Changing Loads on the Nodes of the Network

A distributed system consisting of interconnected workstations experiences a wide fluctuation of loads on individual nodes. A considerable amount of unused computing capacity is always present in the network. Figure 3.1 shows a trace of average loads on four workstations that we obtained in our environment which is predominantly academic. *Vanavil* and *green* are IBM RS/6000 workstations. *Vanavil* is a busy openly used machine with 48 user terminals, whereas *green* is managed by a load leveller that maintains a steady flow of batch jobs for the consumption of the machine. *Bronto* and *elasmo* are two Sun Sparcs which are situated in a smaller laboratory. It can be observed from Fig. 3.1 that the load pattern varies drastically from time to time depending upon the specific user characteristics. It is a challenging task to execute parallel programs in such an environment while achieving a fair amount of load balance. A program is said to be 'load-adaptive' if it adapts to the changing load in the system. However a programmer may not be able to use the knowledge of load fluctuation in the program unless an adequate language support is provided.

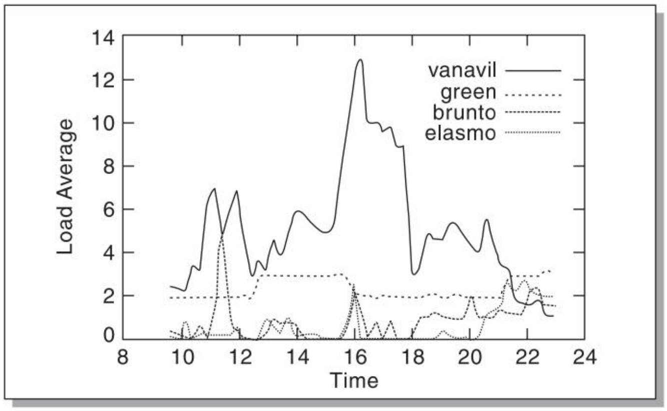


FIG. 3.1

Load Variation on Four Machines in an Academic Environment

The traditional COP model in which the programmer explicitly starts processes on named nodes will be counter-productive in this case. This is because of the fact that a programmer will not be able to identify the changing load patterns in the network. The named node where the programmer starts the processes can get heavily loaded during the course of execution of the program, thereby degrading the performance instead of improving it. Some systems such as PVM (Parallel Virtual Machine) can delay the decision of selecting a node for a process until execution time, but keep this mapping rigid till the process completes its execution. If a load on a particular machine increases during execution, PVM processes on that node would suffer.

3.2.2 Changing Node Availability on the Network

Distributed systems are characterized by nodes that keep going down and coming up over a period of time. Ideally, the computation should not get affected by the changing node availability in the network. For example, a program at a given instance of execution may have made use of five nodes while in another instance, it may have only three nodes available. It is also possible that node failures are transient with respect to one single execution of the program. While computing a sub-task, a node or a link might crash, and may again come up before the completion of the execution of the program.

3.2.3 Differences in Processor Speeds and Network Speeds

Distributed systems consisting of interconnected workstations are characterized by differences in processor speeds and network speeds. The communication overhead in this class of distributed systems is fairly high. Hence these systems are suitable only for coarse grain parallelism. Communication overheads play a major role in influencing speed-ups of parallel tasks on these systems. For example, an investigation on the Apollo Domain network reveals the effect of communication overheads on the speed-up for an RPC-based system [32]. In a heterogeneous cluster, processors have different speeds. However, during runtime, the effective capacity may vary due to loading effects. Selecting appropriate grain sizes during runtime becomes important as a consequence of this variation. Thus, the programming model aimed at interconnected workstations should provide for easy variation of the task grain size.

3.2.4 Heterogeneity in Architecture and Operating Systems

Interconnected workstations normally display some amount of heterogeneity in operating systems and architectures. Heterogeneity in operating systems is being bridged through distributed operating systems, distributed file systems, and standardization of software. Distributed file systems such as NFS are common in the inter-connected workstation environment in order to make the file system available over the network. The architectural differences between workstations are more difficult to handle than operating system differences. Since the binary executable files are not compatible between architectures, it will not be possible to directly execute the binaries relating to a task on the other nodes.

The issues discussed above pose several difficulties in parallel programming on workstation clusters. The management of distribution at the programming level further adds to these difficulties. The existing systems address only a subset of these issues. Providing a single approach addressing all the issues, and at the same time, keeping parallel systems simple to program and to understand, is a real challenge. Most of the current parallel programming approaches for workstation systems follow the COP model that communicates via various messages passing or shared memory abstractions. The COP model gives rise to many difficulties while programming for variations in loads and speeds, heterogeneity, and fault tolerance. The following section provides an overview of the existing approaches, analysing their suitability for parallel programming on distributed workstation systems.

3.3 Existing Distributed Programming Approaches

The COP approaches to programming on workstation systems can be grouped under the following categories with some amount of overlap between the various categories.

- Bare socket programming,
- Remote procedure calls,
- Remote execution,
- Message-passing abstractions,
- Shared memory abstractions,
- Object-oriented abstractions.

Some of them have been proposed for parallel programming whereas others are intended for providing distributed services. We analyse the suitability of these approaches from the parallel programming point of view.

3.3.1 Socket Programming

The most primitive constructs for inter-process communication between workstations are sockets. Communication is at the level of untyped byte streams. This approach provides no support for the requirements discussed in almost all the four categories. Support for intermediate representation such as XDRs is provided in order to handle the heterogeneity of data representation. However, issues such as the selection of nodes, tackling of failures, load adaptability, and heterogeneity at the level of the executable code, have to be handled explicitly by the program.

3.3.2 Remote Procedure Calls (RPCs)

An RPC attempts to bridge the semantic gap between a local call and a remote call. Communication is typed and the services are predetermined. Usually, parallel programs are written with RPC in combination with lightweight threads [8], [32]. There have been attempts to extend the RPC mechanism to heterogeneous environments [10], [42]. These attempts largely focus on the issue of the design of the stub generator for heterogeneous RPC systems. However, these constructs do not address the specific issues in parallel programming on loosely coupled distributed systems. RPC requires services to be placed on servers and exported before a program starts executing. A programmer can explicitly start processes on nodes, which might lead to severe load imbalances in the system. In other words, these constructs do not make a distinction between the programmer's concerns and the system's concerns in the case of loosely coupled distributed systems. RPC is more appropriate for providing distributed services rather than for writing parallel programs on workstation clusters.

3.3.3 Remote Execution

A remote execution facility allows for migrating code fragments to remote servers for execution. The REV has been developed for providing migration of executable code to servers [37]. The main motivation in developing REV is to reduce the client-server communication. Instead of moving large amounts of data from servers to clients for processing, the executable code is migrated to the server site and the results are sent back to the client. A similar approach can also be found in computation migration [23]. Remote execution as a standalone paradigm is also not suitable for parallel programming on workstations due to its lack of adequate support for heterogeneity, load adaptability, and fault tolerance.

3.3.4 Message Passing Abstractions

These are the higher level abstractions to bare socket programming. The concept of typed data communication is introduced and untyped byte communication can still be retained. Programming languages such as Lynx [35] and programming platforms such as PVM [40], P4 [11], and MPI [29] support various abstractions of message-passing. For example, PVM supports the transmission of several data types such as integer or float streams along with byte streams. Some of the problems faced by the bare socket programming model are eliminated here. The model is easier to use due to its elegant abstractions. However, adequate support is not provided for heterogeneity, migration, load adaptability and fault tolerance. If a remote machine fails, the state of a process on that machine is lost and such a failure has to be handled explicitly at the programmer's level. Message Parallel Virtual Machine (MPVM) [14], a version of PVM, provides support for process migration. However, process migration-based solutions are useful for adapting to load variations on homogeneous systems only.

3.3.5 Distributed Shared Memory Abstractions

They range from bare distributed shared memory support such as the Mether system [28], TreadMarks [2], and Munin [13] to high-level shared memory abstractions such as Orca [7] and Linda [12]. Shared memory abstractions appear to provide a much easier programming interface than message-passing. Linda supports distributed data structures to which multiple processes can have simultaneous access. Multiple processes can be started on different nodes. In Orca, objects are shared between processes and their children. A new process can be forked explicitly on a processor by using a 'fork' construct. Once a process is forked on a machine, it may suffer from loading effects. Once the number of processes is fixed by a program, it cannot utilize the additional computing power that becomes available during runtime.

3.3.6 Object-oriented Abstractions

In recent years, several object-based and object-oriented paradigms such as Emerald [9], Orca [7], CHARM++ [27], and Mentat [18] have been proposed for distributed computing. Processes are replaced by objects. The communication in this case is inter-object rather than inter-process. Consequently, the COP model is still preserved but at the level of objects. The various drawbacks of these abstractions in the context of workstation systems have been discussed in [26].

In this context, we introduce the paradigm of ARC that eliminates the problems of the COP model of parallel programming on workstations. Programs are not written as a collection of communicating processes, but as a collection of several loosely coupled RIBs within a single program entity. RIBs are open-ended and migrate to convenient anonymous remote nodes during runtime. This approach makes programs fault-tolerant, heterogeneous and load-adaptable. At the same time, ARC keeps programs easy to understand by separating the system's concerns from the programmer's concerns.

For the programmer, the nodes in the network remain anonymous and a runtime system starts the required processes on remote nodes by considering the loads on these nodes. In order to handle architectural differences, the source code of tasks that are RIBs, is identified and stored separately. The mechanisms of migration and lazy compilation of RIBs are used for executing the tasks on heterogeneous nodes. Primitives are provided for handling variations in loads and speeds dynamically.

3.4 The ARC Model of Computation

The ARC model of computation is designed to meet the following two goals of parallel programming on workstation systems:

1. Clear separation between programmer's concerns and system's concerns, and
2. Account for heterogeneity, fault tolerance, load adaptability, and processor availability.

The origin of the conventional COP model of parallel programming on workstation systems can be traced to the process-based multi-programmed machine with various inter-process communication mechanisms. Networking of multiple multi-programmed machines led to a direct adaptation of this model to distribute programming as in Synchronizing Resources (SR) [5] and Lynx [35], and subsequently to parallel programming as in PVM [40]. The trends in the cluster computers show proliferation of a large number of heterogeneous workstations in a Local Area Network (LAN) configuration. An important characteristic of this cluster of workstations is the dynamically changing loads on them, which makes the task of harnessing the idle computing capacity for parallel programming a very challenging task. The changing load on these systems coupled with link and processor failures make the conventional COP model unsuitable for parallel programming on these systems. The ARC model has been proposed to address these requirements of parallel programming on a cluster of workstations.

Since multiple processes are involved in the COP model, programs are difficult to understand and debug in it. The explicit programming of communication between entities may lead to synchronization errors. The ARC model aims at eliminating the creation of explicit entities mapped onto different machines. A program in the ARC model consists of a single entity comprising multiple loosely coupled blocks.

Figure 3.2 shows an overview of the ARC model. An ARC program consists of several loosely coupled RIBs linked via synchronizers. An RIB is a code fragment that can be executed on an anonymous remote machine. An ARC program may have multiple RIBs that can be executing simultaneously. RIBs are synchronized with the help of synchronizers. RIBs do not involve mechanisms of process creation, or inter-task communication. The nodes on which the RIBs have to execute remain anonymous to

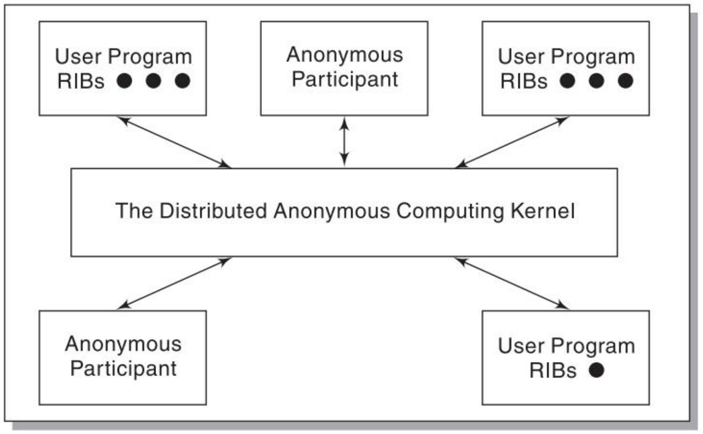


FIG. 3.2

Overview of the ARC Model

the program. The runtime system known as the ARC system decides the nodes on which RIBs are to be executed. The target node thus remains anonymous to the program. It is also possible to migrate an RIB onto a heterogeneous node. An RIB is submitted for a possible remote execution by a user program. At a time, multiple programs can be generating RIBs and multiple anonymous remote participants can be joining or leaving the ARC system. An RIB can further generate new RIBs.

3.4.1 Comparing ARC and COP Models

The computing entities in the COP model are processes or objects working directly on top of the operating system, whereas in the ARC model, they are remote instruction blocks within a single program. Figure 3.3 brings out the differences between the current COP approaches and the ARC model on the basis of patterns of synchronization among the computing entities. The COP approaches are classified into five classes, viz. task synchronous, call synchronous, call asynchronous, message synchronous, and message asynchronous systems.

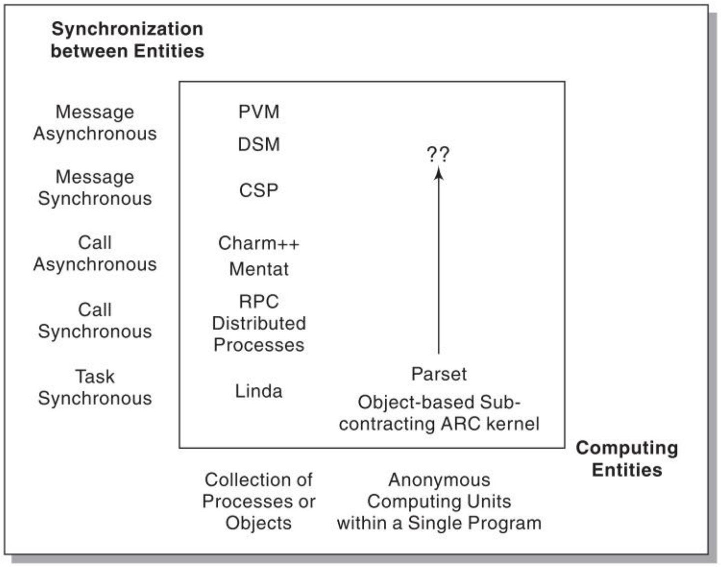


FIG. 3.3

Comparing COP and ARC Models

3.4.1.1 Task Synchronous Systems

A task synchronous system provides synchronization between computing entities at the task level. Two computing entities are synchronized at initiation time or the completion time of tasks. The Linda [12] system is an example of the task synchronous COP approach. Tasks can be deposited in a tuple space and decoupled processes pick up the deposited tasks.

3.4.1.2 Call Synchronous Systems

In a call synchronous system, two computing entities communicate by calling methods defined by each other. When an entity calls a method, it expects an action to be taken and further blocks for the result to return. RPC [31] and the Distributed Processes model [21] are the examples of call synchronous COP systems.

3.4.1.3 *Call Asynchronous Systems*

Some of the distributed object-oriented programming systems are call asynchronous systems. In these systems, method calls return immediately in a non-blocking fashion. Either a 'future' mechanism is used or a 'call back mechanism' is implemented to obtain the return value of a call. The 'latency tolerance' mechanism of CHARM++ [27] is an example of a call back system whereas the 'return-to-future' mechanism of Mentat [18] is an example of a future-based call asynchronous system.

3.4.1.4 *Message Synchronous Systems*

Two computing entities exchange messages such as arbitrary data values. Message communication is blocking. Communicating Sequential Processes (CSP) [22] is an example of a message synchronous COP system.

3.4.1.5 *Message Asynchronous Systems*

Two entities exchange messages such as data values at any arbitrary point of time. This type of synchronization can be achieved by shared memory as well as message-passing techniques. Since a message can arrive at any time, messages are usually typed and the receiving process knows the type of the incoming message. The conventional Distributed Shared Memories (DSMs) and message-passing mechanisms such as PVM [40] and MPI [29] are the examples of a message asynchronous COP system.

3.4.1.6 *The ARC Approaches*

ARC extends the computing entities in the horizontal domain to RIBs. However, in the vertical domain, ARC can be classified as a task synchronous system. In an ARC system, the synchronization is provided at the beginning or completion of RIBs that can be seen as loosely coupled tasks. The synchronization is provided with the help of synchronizers as discussed later in this section. Parsets [25] and Object-based Sub-contracting (OBS) [26] are two high-level ARC approaches proposed by us that fall into this category.

The development of message synchronous and message asynchronous procedural ARC paradigms, and call synchronous and call asynchronous object-oriented ARC paradigms requires support for communication among RIBs executing on anonymous nodes. This is a very challenging task and needs further research. The arrowhead in Fig. 3.3 captures the scope for future developments in ARC paradigms.

3.4.2 RIBs and Synchronizers

Figure 3.4 shows an example adaptive quadrature [4] program depicting the use of RIBs and synchronizers. RIBs are the code segments that can be executed at anonymous nodes. Synchronization between independently executing RIBs is achieved through synchronizers.

3.4.2.1 Remote Instruction Blocks

In Fig. 3.4, a special variable called RIB variable *R* is introduced to mark a control specification as a remotely executable block. The variable *R* is bound to a function call *AQ()* that computes the area under a curve bounded by given left and right boundaries. Once an RIB variable is bound, a reference to the variable in an EXEC call executes the code associated with the RIB variable. The EXEC call encapsulates the system's concerns such as fault tolerance, heterogeneity and load adaptability from the program. An RIB is migrated to a suitable remote node by the runtime system. In the case of heterogeneity, the RIB source code is migrated and the mechanism of lazy compilation is used to execute the RIB.

The approach of the migrating code was exploited earlier by researchers in the case of functional languages. Falcone developed a LISP-based language called NCL to support Remote Function Evaluation (RFE) [19]. In a different approach, Stamos and Gifford developed a remote form of evaluation [37], which introduces the execution of a function at a remote node. In contrast to these approaches, RIBs are open-ended blocks for which the target machine for execution is unknown until execution time. An RIB execution isolates the programmer from the addressing system's concerns at the programmer's level.

```
main () {
...
double left, right, middle;
double area1, area2, totalArea;
RIB R;
SYNC S;
...
middle = (left + right)/2;
                //divide the task
R area1 = AQ (left, middle);
                // mark one task as RIB
S = EXEC R;     // execute RIB
area2 = AQ (middle, right);
                // inlined code of second
                   task
WAITFOR S;      // synchronizer
totalArea = Area1 + area2;
...
}
```

FIG. 3.4

An Example of an ARC Code

An RIB should support the following two basic properties:

1. Open-endedness,
2. Fault tolerance.

Open-endedness

The target machine for execution is unspecified. As a consequence of this property, heterogeneity and load adaptability are supported. Since the target machine is unspecified, a suitable anonymous node may be selected at runtime depending upon the load conditions. If the anonymous node is heterogeneous, the source code of the RIB needs to be migrated.

Fault Tolerance

Support is provided to make sure that an RIB is executed at least once. In case of the failure of an anonymous node, the RIB may be executed on a different node, or in the worst case, execution on the host node can be guaranteed. The choice of a scheme of fault tolerance depends upon the semantics of the high-level ARC constructs. For example, an ARC construct may provide automatic execution on the host node by in-lining the failed unit in the case of repeated failures, whereas another ARC construct may report the failure to the program and leave the decision of re-executing the code fragment to the program.

ARC is developed in a two-tiered architecture. At the top layer, RIBs are expressed in a high-level program by using language extensions to existing languages such as C or C++. At the lower level, the basic primitives for anonymous remote computing are provided. It is possible to develop different higher level paradigms that provide the above two basic characteristics of RIBs. We have earlier proposed two high-level ARC paradigms, Parsets and OBS, which are tailored to the needs of procedural and object-oriented programming, respectively. These two paradigms extract RIBs from specially designed language constructs.

3.4.2.2 Synchronizers

Synchronizers are mechanisms that are needed to achieve synchronization between independently executing RIBs. Two types of synchronizers called 'Sync variables' and 'lock specifications' are provided. Figure 3.4 shows the first type of synchronizer, the SYNC variable. An RIB execution is assigned to a SYNC variable, *S*. At any later point, the program can be made to wait on the SYNC variable to assure that the execution sequence of the associated RIB is completed. Functional languages have used similar variables called future variables [20].

Parsets and OBS use lock specifications as synchronizers. In Parsets, function arguments are locked as RO, WO, or RW during execution. If an argument is RO-locked, it can be made available to a subsequent function execution. A function begins its execution when desired locks on all of its arguments are obtained. A similar scheme is used in OBS wherein locks are obtained on objects for message invocations. Locks provide auto-synchronization whereas SYNC variables are explicit synchronizers.

3.4.3 Dynamically Growing and Shrinking Trees

Computationally, ARC programs can be represented as 'dynamically growing and shrinking trees' as shown in Fig. 3.5. An RIB can further generate new RIBs. As RIBs complete execution, the tree shrinks.

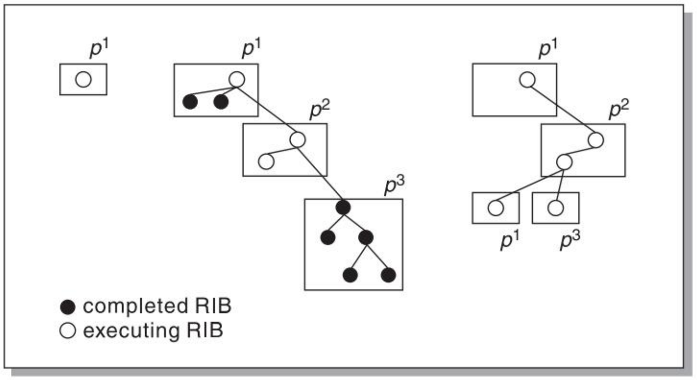


FIG. 3.5

Dynamically Growing and Shrinking RIBs

If more RIBs are generated during execution, a shrinking tree may again start growing. Since ARC is a task synchronous system, synchronization is provided at the beginning or completion of an RIB. Communication between RIBs is not supported. The capability of RIBs to spawn new RIBs along with the task synchronization scheme results in the dynamically growing and shrinking tree model. Achieving different types of RIB communication rather than just task synchronization is a very challenging task and needs further research.

3.4.4 Benefits of the ARC Model

The ARC model is based on RIBs contained in a program instead of a COP that communicates explicitly. This approach grants distribution transparency to parallel programs on workstations. RIBs can be made heterogeneous by the ARC system by runtime source-code migration and re-compilation. In this way, ARC also provide heterogeniety transparency. Transparency to distribution and heterogeneity removes the burden of programming for distribution and handling of heterogeneity at the programmer's level. The ARC approach grants the property of statelessness to a remote server executing a parallel task. Since ARC avoids the notion of processes containing state of parallel computation, it does not face the problems that arise in the COP model out of the failures of its participant processes. Stateless executors and the ability to locally or remotely re-execute RIBs free the programmer from the task of handling failures at the programming level. The ARC model makes programs load-adaptable. The dynamically growing nodes of a task tree may be suitably clustered and awarded to available machines. Transparency to distribution and heterogeneity makes it possible to load the tasks dynamically on suitable machines. Moreover, ARC provides an abstraction called Horse Power Factor (HPF), which gives a measure of the current processing power of a machine. Abstracting load and speed (i.e. heterogeneity) in this way makes it possible for a parallel program to execute on unevenly loaded heterogeneous machines.

3.5 The Two-tiered ARC Language Constructs

In this section, we introduce the two-tiered ARC language constructs. The lower layer provides primitives to support system tasks such as the creation of RIBs, fault tolerance, load sensing and source code migration. The upper layer consists of higher level language extensions that advocate ARC programming methodology.

3.5.1 The Evolution of RIB

Execution of the code at remote nodes is a widely used programming paradigm in distributed systems. As classified in Section 3.3, they fall into two categories of RPCs and remote execution. Table 3.1 summarizes the main differences between the RIB approach and these paradigms.

RPC is perhaps the most popular remote execution paradigm used to build distributed applications. Many specific RPC protocols have been developed in recent years to meet the application-specific requirements. For example, the blocking RPC [31] has been extended to MultiRPC, in which, the client can make calls simultaneously to multiple servers for locating files [34]. In the conventional RPC paradigm, the remote procedure is compiled and registered with the server process. In other words, the server exports a set of procedures that can be invoked by the clients. In the case of ARC, a program identifies several RIBs that may be executed on remote nodes concurrently. Thus the RIBs are part of client programs and not of the procedures that servers export. RIBs can be migrated and executed on other nodes.

Migration of RIBs is different from the heavyweight process migration employed by operating systems for load balancing [1], [36]. However, a programmer may not distinctly benefit from process migration mechanisms unless the program is broken into a collection of concurrent processes as in the COP model.

REV has been developed for providing the migration of executable code to servers [37]. The main motivation in developing REV is to reduce the client-server communication. Instead of moving large amounts of data from servers to clients for processing, the executable code is migrated to the server site and the results are sent back to the client. A similar approach can also be found in computation migration [23]. The computation is migrated to the data site to exploit the locality of data references. Although our approach resembles the REV and the computation migration in migrating the code to the remote node, the motivation is parallel programming on anonymous nodes providing fault tolerance, heterogeneity and load adaptability rather than distributed application development with well-known servers and clients.

Table 3.1 The Evolution of RIB

Mechanism	*Motivation*	*Execution*	*Remarks*
Process Migration	Load balancing	Runtime migration of processes	Heavy migration cost, unsuitable for heterogeneous architectures
Computation migration	Enhancing locality of references	Migrate computation accessing remote data	Does not address parallelism
RPC	Transparent access to remote procedures	Blocks the client when server executes the call	Placement of functions on servers is predetermined, parallelism not supported
muti-RPC	Multiple remote calls at a time	Multiple RPC calls made to servers	Used in locating files from multiple servers, function placement is predetermined
REV	To reduce client-server communication	Function migrates from client to server at runtime	Parallelism is not addressed, focus is on distributed system services, program is given control over location of processing
RIB	To support ARC paradigm, to integrate fault tolerance, load adaptability, scalability, heterogeneity from parallelism point of view at language level	Procedures migrated at runtime, servers are anonymous	Supports anonymous computing on workstation systems, nested RIBs possible, runtime as well as compile time support is required

RIBs are the basic building blocks of an ARC program. The lower layer ARC interface consists of a set of constructs which are used to specify and manipulate various parameters associated with RIBs. The distributed ARC kernel provides the runtime support for the RIBs. An RIB can be submitted, executed, and manipulated with the help of the ARC kernel interface calls.

3.5.2 The Design of the Lower Layer ARC Interface

The lower layer ARC provides support for the execution of RIBs, fault tolerance and load adaptability. An ARC-distributed kernel implements these functionalities. The implementation of the kernel is described in the next section while the ARC interface is described in detail in this section. In this sub-section, a detailed overview of the ARC Primitives (ARCPs) is provided.

An RIB execution goes through the following steps:

1. obtain a lock on an anonymous computing unit,
2. pack the arguments to the RIB,
3. post the RIB on the lock obtained,
4. obtain the results of the RIB execution.

Locks have to be obtained on anonymous remote nodes as a confirmation for execution. The kernel releases locks depending upon the number of anonymous computing units in the systems. Locks are associated with an HPF. The HPF encodes the load and processing capacity of an anonymous node. After obtaining a lock, an ARC program may dynamically post a larger or a smaller RIB grain depending upon the lock value. After securing a lock, the arguments to RIB are packed together and the RIB is posted to the kernel. The results of an earlier posted RIB can be obtained at a later point of time. The ARCPs are discussed in three groups namely, start-up and close-down primitives, RIB support primitives and the HPF and asynchronous intimation primitives for load adaptability.

3.5.2.1 Start-up and Close-down Primitives

Before generating the ARC calls, a program needs to identify itself with the ARC kernel. This is done by using the *ARC start-up()* primitive. A call to this primitive establishes the necessary connections and performs the necessary initializations. The kernel keeps track of open connections. The *ARC close-down()* primitive unlinks the program from the ARC kernel, disconnects already established connections, and advises the kernel to update its state.

3.5.2.2 Primitive Support for RIBs

This section explains the RIB primitives for packaging arguments, posting RIBs and receiving remote results. RIBs are extracted from an ARC program and kept separately in the local file system. They are made available in the form of compiled code as well as in the form of source code. Depending upon the location of the remote machine and its heterogeneous or homogeneous nature, the source code or the executable code is migrated.

Primitives for Packaging

The *openDataPack()* primitive returns a new data-pack handler, in which an arbitrary number of arguments can be inserted by using the *insertDataPack()* primitive. Data is inserted into the data-pack pointed by the data-pack handler. Any number of insert calls can be made on a particular data-pack handler till the handler is closed for further inserts. A contiguous packet is prepared for all the inserted arguments by calling the primitive *closeDataPack()*. Every RIB prepares its new data-packet with these three primitives.

Primitives for RIB Posting

The *postRib()* primitive is used to post an RIB to the ARC kernel which can allocate the RIB to an earlier obtained lock. It is a non-blocking call, and it returns immediately with a work identifier. The work identifier can be used at a later point of time to obtain its results and also to alter the parameters related to the corresponding RIB. Its arguments are the RIB identifier, the argument packet handler, and parameters to control re-execution.

If the remote node fails, the *retries* parameter can suggest re-execution on a possibly available anonymous node if the *time-out* for the RIB is not elapsed. The *retries* parameter gives the maximum number of retries to be made within the given *time-out* for the RIB. A specific fault-tolerant behaviour can be obtained with the help of this *retry-till-time-out* mechanism. These two parameters can be changed dynamically as explained subsequently.

The primitive *obtainResult()* is used to obtain the results of an earlier RIB posted for remote execution. It returns a structure containing a status value along with other information such as the actual result pointer and execution time for the successful execution.

The status value *WORK_IN_PROGRESS* is returned if the time-out is reached but the result is not yet available. In this case, the time-out value may be incremented with the help of a call to *setParam()*. If the time-out is not reached, the call blocks. Subsequently, if the RIB is successfully executed, the call returns with a status value *RESULT_OBTAINED* along with the actual result. In case of a failure of the remote node, if the time-out value for the RIB is not crossed and retries are warranted, it tries to obtain a lock on an anonymous node to resend the RIB for re-execution. In case of a successful resend, the call returns with a value *FRESH_TRY*. If an anonymous node cannot be contacted due to its unavailability or due to the time-out value, a value *REM_FAILURE* is returned. In this case, the program is advised to execute the code by in-lining or to try again at a later time for a remote execution.

Primitives for Parameter Setting

The *setParam()* primitive is used to set the control parameters for a particular RIB posting. The parameters that can be controlled are the time-out value and the number of maximum retries that can be made within the time-out period in the case of failures. The time count starts when the work is posted. In the case of failures, the ARC kernel internally manages a fresh execution of the ARC call as explained above. The values of the parameters can be altered dynamically. A fresh call to *setParam()* resets the parameters to new values. Time-out can be set to an arbitrarily large value of INFINITY.

3.5.2.3 Horse Power Factor and Asynchronous Intimations

Load adaptability is a major concern for ARC. ARC provides basic primitives to provide load adaptability to programs without enforcing the actual load distribution scheme at the kernel implementation level in contrast to earlier schemes such as the multi-level scheduling in CHORES runtime system [17]. HPF and asynchronous intimations are the two primitives that help programs to become adaptable to variations in loads and speeds of machines in the clusters.

The Horse Power Factor (HPF) Primitive

An HPF corresponding to a machine at a given time represents its processing capability at that time. It is a dynamically varying number. Various machines in the network are normalized by a benchmark program to obtain a relative index of speeds. This relative index is statically obtained. The HPF for a machine is dynamically obtained by considering this relative index and the load on that machine. The motivation behind the HPF is that, if differently loaded heterogeneous machines generate the same HPF figure, a task awarded to them is ideally expected to take the same time. Table 3.2 captures the utility of the HPF. The task run was a double precision 200×200 matrix multiplication program. The test was conducted on load conditions. The task was sub-divided into four smaller tasks based on the HPF obtained. It can be seen that a fair amount of load balance is obtained. More detailed studies for dynamically generated tasks are presented in the Performance section.

The obtainLock() primitive is used to secure a lock on an anonymous node and obtain the HPF for the locked unit in return. Using the HPF, the program may post an RIB with an appropriate grain size. The call takes an input argument that specifies an expected processing power. The expected processing power can be chosen from the available standard values of LOW, MEDIUM, HIGH, XHIGH, and ANY.

Table 3.2 **Performance of the Horse Power Factor**

Task: 200 × 200 Matrix Multiplication				
Machine	*No. of users on the machine*	*HPF observed*	*Allocated task size*	*Execution time (sec)*
Pelcan (Sun 3/50)	2	72	2 × 200	10.76
Bunto (Sun Sparc)	4	853	18 × 200	9.83
Pterano (Sun Sparc)	9	1457	30 × 200	8.29
Vanavill (IBM RS/6000)	8	7420	150 × 200	10.87

The Asynchronous Intimation Primitive

The asynchronous intimation facility is provided in order to support dynamically growing tasks. With this facility, whenever an anonymous node becomes ready to accept remote RIB requests, an intimation can be delivered to an ARC-registered program. Such an intimation is only an indication of a possible availability and not a guaranteed allocation. A lock can be obtained upon the delivery of an asynchronous intimation.

A call to the *intimationReceived()* primitive returns a value TRUE if an asynchronous intimation is received earlier. Any subsequent call to this function requires the receipt of a new intimation in order to obtain a value TRUE again. Asynchronous intimations help in reducing lock-seek time, since every lock-seek operation involves contacting the ARC-distributed kernel whereas asynchronous intimations are accessible with a function call. If the call returns a value FALSE, the lock-seek overhead can be avoided. This facility is especially useful for dynamically growing tasks. In such cases, upon the receipt of an asynchronous intimation, a lock can be tried and subsequently, a new task can be posted.

3.5.3 The Upper Layer ARC Constructs

The primitives explained above are built over the ARC kernel. A high-level ARC language paradigm may use these primitives in various ways to provide easy-to-use ARC language constructs. We have earlier proposed Parsets and OBS, the two ARC language paradigms for procedural and object-oriented programming, respectively. Although these paradigms use a specific implementation different from the one described above, they are good examples of high-level ARC paradigms. They use argument locking and object locking for synchronization. RIBs are function calls in the case of Parsets and meta message invocations called sub-contracts in the case of OBS.

We present another example of a high-level ARC construct, the ARC function call, in the following section. While Parsets and OBS capture parallelism *within* a single method invocation or a single sub-contract, the ARC function call models a sequential function invocation which may be executed at an anonymous remote node. An ARC function call can be made blocking or non-blocking. A non-blocking ARC function call is provided with a synchronizer in order to support concurrency *outside* the ARC call.

3.5.3.1 *Blocking and Non-blocking ARC Calls*

In this scheme, the user can tag the functions as belonging to one of the blocking or the non-blocking call classes. Examples for these calls are provided in Fig. 3.6. While generating the lower level ARC interface calls from these specifications, appropriate default values are set for *time-out* and *retries* parameters. The blocking adaptive quadrature call may be executed on a remote anonymous node and the call returns only after successful completion. Internally, the runtime environment should manage the re-execution or in-lining in the worst case. Default values for *time-out* and *retries* can be chosen as INFINITY and ZERO. The blocking ARC function call provides functionality similar to RPC, with the difference of the client migrating the function to be executed on an anonymous node. Instead of running a function on a slow host, it can be migrated transparently to a fast node. The blocking ARC function call is not designed for providing parallelism, but is targeted at providing speed-up to a conventional sequential function call.

The non-blocking version of the ARC function call needs a synchronizer. While the call returns a synchronizer, it may be

```
double  AQ-blocking  (double  lower,  double
upper)
      << BLOCKING ARC >> {
      double area;
        //  compute  area  under  the  curve
        with given cut offs
      return (area);
      }
    SYNC AQ-nonBlocking (double lower, double
    upper)  : double
    << NONBLOCKING ARC >> {
      double area;
        //  compute  area  under  the  curve
        with given cut offs
      return (area);
    }
    main () {
      double L, U, Area;
      SYNC S; ...
      S = AQ-nonBlocking (L, U); // function
      call returns immediately
      ...
      Area = AQ-blocking (L, U); // blocks
      till completion
      print  Area;  //  return  value  of  the
      blocking function
      waitOn (S, &Area); // synchronizer
      print  Area;  //  return  value  of  the
      nonblocking function
    }
```

FIG. 3.6

Blocking and Non-blocking ARC Function Calls

executing on a remote anonymous node. At a later point of time, the result of the non-blocking call can be obtained by waiting on the corresponding synchronizer. It can be seen that the synchronization scheme in Fig. 3.6 differs in a minor way from the scheme given in Fig. 3.4. In the example shown in Fig. 3.4, the high-level ARC construct makes only a particular function invocation as an ARC call by using RIB variables, whereas, in this example, all invocations of a given function are treated as ARC calls. The *postRib()* function call described above is an example of a lower level non-blocking ARC function call. In this way, it is possible to provide appropriate higher level ARC language semantics to suit the higher level requirements. More detailed higher level ARC language constructs are discussed elsewhere [25], [26].

3.6 Implementation

The anonymous remote computing mechanism is provided as an extension to C language. It is implemented on a heterogeneous cluster of workstations consisting of IBM RS/6000s, Sun Sparcs and Sun 3s. There are four IBM RS/6000 running AIX, three Sparcs running Solaris 2.3, and 18 Sun-3s running SunOS 4.0.3. In this section, we describe the implementation of ARC in this environment.

Figure 3.7 depicts the architecture of the ARC implementation. A distributed ARC kernel is spread over the workstations that participate in anonymous remote computing. The entire domain is partitioned into three logical clusters having a separate file system for each one. The ARC kernel consists of multiple local coordinators to co-ordinate local activities and one system coordinator to co-ordinate the global activity. Each machine that intends to participate in the ARC system runs a local coordinator. The kernel supports the lower layer ARC primitives. It is possible to write ARC programs by directly using these primitives. Upper layer ARC programs such as those discussed in earlier sections can be converted into lower layer ARC programs consisting of these primitive calls.

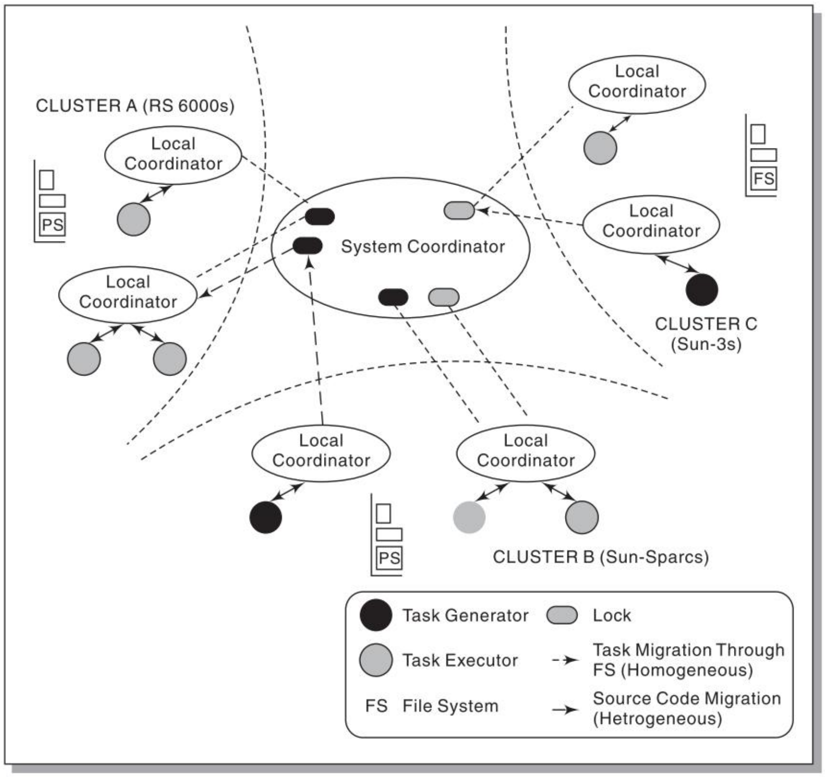


FIG. 3.7

The Distributed ARC Kernel

3.6.1 The System Coordinator

There is only a single system coordinator in a given domain of logically grouped clusters. The system coordinator's functions are to manage locks, route RIBs, and maintain migration history. It only functions as a policeman controlling and routing the traffic of ARC calls. It does not function as a task heap. In the following sub-sections, we discuss the various functions of the system coordinator.

Lock Management

Whenever a machine wants to improve its utilization, it registers with the system coordinator through its local coordinator. The system coordinator then creates a free lock for this machine. A machine may deposit any number of free locks with the system coordinator through its local coordinator in order to boost its utilization. A 'wait queue' is built for local coordinators seeking external RIBs through these free locks. Whenever a lock request arrives, a statistics-daemon is contacted to access the current load information for a given machine. From the current load information and the normalized speed of the machine, the HPF is computed and returned in response to the lock request.

Routing RIBs

An RIB posting is routed to the corresponding free local coordinator for execution. An RIB posting consists of the argument data-packet, the source code or compiled code, whichever is appropriate, and the make directives for various architectures. In case of heterogeneity, the RIB source code is requested. Along with the source, the relevant make directives are also supplied. In our implementation, all homogeneous machines have access to RIB binaries through the file system, whereas, across heterogeneous machines, the source code is migrated and RIB binaries are prepared for the target machine. The results of the RIBs are sent back to the corresponding local coordinators.

Maintaining Migration History

A history of recent migrations is maintained. In the case of dynamically growing tasks, a machine may receive an RIB task belonging to an earlier posted source code. If an RIB source is already posted to a machine in a heterogeneous cluster, the binaries available in the file system of that cluster can be accessed directly. Thus, if an RIB with the same source arrives later for any machine in that cluster, repetitive migration and recompilation are avoided by using the history of recent migrations across clusters.

3.6.2 The Local Coordinator

The local coordinator runs on a machine that participates in the ARC system either to improve its utilization or to share its work with an anonymous remote node. Any ARC communication to or from the local processes is achieved through the local coordinator. The tasks of the local coordinator are described in the following sub-sections.

Improve Local Utilization

An online command can be executed that directs the local coordinator to obtain a particular amount of work from the ARC system. This generates an asynchronous intimation that travels to the remote programs which may possibly use them to execute newly generated RIBs. When an RIB arrives, the local coordinator creates a task executor process corresponding to the RIB. Communication between a task executor and the coordinator is implemented by using Unix Domain sockets. The task executor process receives the RIB arguments and executes the RIB code. The results are forwarded to the local coordinator. In the case of a dynamically growing task, the task executor is retained and another set of arguments for the same RIB may arrive. A task executor may also generate new RIBs. These RIBs can be further submitted for remote execution.

Accept RIBs from Local Processes

RIBs are initially generated by user processes. Later the RIBs themselves may generate new RIBs. A new RIB may be generated upon the receipt of an asynchronous intimation. The local coordinator provides the intimation to a program by using a signal. When the results of remote RIBs arrive, they are queued up in a result queue. When a synchronizer in the program requires a particular result, the result queue is searched.

3.6.3 The RIBs

RIBs are extracted from the program by using an ARC compiler. Following are the major tasks of the ARC compiler. An RIB code is generated by using a generic RIB process that accepts arbitrary arguments and posts a result. Dynamic RIBs do not die immediately and can further accept a new set of arguments. A specialized RIB is prepared from the generic RIB by integrating the required code fragments into it. Also associated with each RIB is a set of make directives that dictate the compilation process on different available architectures. Whenever an RIB is migrated to a heterogeneous architecture, the appropriate make directive is also migrated. Using the make directive, the remote local coordinator compiles the RIB.

3.6.4 Time-outs and Fault Tolerance

The code that maintains the time-outs is integrated with the code that generates RIBs. A timer signal ticks at regular intervals. The *time-out* parameter in the lower layer ARC primitives is specified in terms of this interval. The current clock value is maintained in a variable that is accessed by the *obtainResult()* primitive to provide the time-out semantics as explained in the earlier section.

A remote node failure is detected by the system coordinator when the local coordinator fails. The system coordinator maintains a list of incomplete jobs allocated to various local coordinators. Whenever the failure of a coordinator is detected, an RIB failure intimation is sent to the source program. A local coordinator may detect the failure of an executing RIB process, which might possibly occur due to program bugs. Such failures of an RIB are also reported back to the source program of the RIB. Upon the detection of a failure, if the values of the *time-out* and *retries* parameters permit a re-execution, the RIB may be resent to another suitable remote node.

3.6.5 Security and Portability

Security in ARC implementation is built over the security provided by operating systems over which ARC is implemented. The system coordinator accepts connections only from local coordinators on trusted clients. The addition of a new machine into the system has to be registered with the system coordinator before it can be used to run ARC programs. A remote local coordinator accepting

RIBs arriving from anonymous nodes runs with specific user permissions especially created for the purpose of ARC computations.

Portability to ARC programs is provided by making various ARC libraries portable across systems that are registered with the ARC system coordinator. Typically, the libraries that support the implementation of ARC primitives are made portable. However, the portability of the user's C code has to be ensured by the user.

When an RIB migrates to a heterogeneous node for dynamic compilation, the directives for the compiling environment are also migrated along with the RIB source code. An example of the compiling environment directive is the *–lm* option for compilation of an RIB that references functions supported by a portable *math.h* package.

3.7 Performance

The ARC approach provides an alternative to the conventional COP approach of parallel programming on a network of workstations. ARC provides for heterogeneity, fault tolerance and load adaptability to parallel programs. In this section, we detail various performance tests to highlight these aspects of ARC.

3.7.1 Adaptability on Loaded Heterogeneous Workstations

This test was conducted to highlight the capability of an ARC program to run effectively on a heterogeneous system in the presence of load. The network consisted of IBM RS/6000s, Sun Sparcs and Sun 3/50s and Sun 3/60s workstations. The test was conducted for a TSP using a variant of SA algorithm described by Janakiram, et al. [24]. This problem was chosen for the adaptability test due to its high computation to communication ratio, and dynamic task generation capabilities. The program starts with an initial solution space and an initial temperature. Locks are obtained on available anonymous processors by dividing the initial solution space among them. Each processor carries out a number of iterations over each grain that it receives. As the results are brought back, the temperature is gradually reduced. As the temperature reduces, the tree is expanded by a factor of 2 to generate tasks dynamically. The tree can have smaller depths while it explores more breadth-wise. Each time a node finishes its task, a fresh lock is deposited by the node in order to obtain a new set of tasks. The ARC program assigns a varying number of tasks on a lock based on its judgment of the corresponding HPF. In every hop, a processor may get a different task size based on its current HPF. Obtaining a lock involves an RPC call by the system coordinator. The system under test consisted of two different networks, an FDDI ring of 100 MB/sec capacity and an Ethernet of 10 MB/sec capacity. The lock time was of the order of 100 msec on load conditions.

The results of the test are summarized in Tables 3.3 and 3.4. The load on the machines consisted of programs such as editing and compiling, simulation experiments generating CPU load, worldwide web (www) browsers generating heavy network traffic, and background batch jobs. We introduce the Horse Power Utilization (HPU) figure to capture the utilization of a loaded heterogeneous multi-programming multi-processor environment. All the timing figures are quoted as the real-time taken. The HPU for p processors and k heterogeneous clusters can be computed as:

$$HPU = \frac{\text{Total remote computation time} + \text{Total remote compilation time}}{P * \text{Total execution time taken by the host program}}$$

$$= \frac{\sum_{i=1}^{P} T_{ci} + \sum_{j=1}^{k} T_{compile\ j}}{pT_t}$$

where

T_t = Total time taken by the host program,

T_{ci} = Computation time taken by ith processor

$T_{compile\ j}$ = Compilation time taken by jth heterogeneous cluster

Table 3.3 The Heterogeneity Load Test, Sixteen Processors

Task A dyanamically generated tree of total 3100 notes with 2000 iterations per node					
Machine	*No. of hops taken*	*Average HPF per hop*	*No. of tree nodes processed*	*Compilation time (sec)*	*Processing time (sec)*
IBM RS/6000	33	9209	831	1.31	514.89
IBM RS/6000	18	5088	441	waiting time	592.65
IBM RS/6000	19	5040	441	waiting time	581.80
IBM RS/6000	18	4916	440	waiting time	584.21
Sun Sparc	16	1127	286	20.55	571.44
Sun Sparc	14	959	220	waiting time	563.60
Sun 3/60	40	54	70	nil	540.92
Sun 3/60	39	53	67	nil	546.70
Sun 3/50	35	36	45	nil	574.08
Sun 3/50	34	36	44	nil	556.98
Sun 3/50	36	36	47	nil	565.92
Sun 3/50	30	35	39	nil	566.92
Sun 3/50	29	32	37	nil	557.10
Sun 3/50	30	32	38	nil	562.56
Sun 3/50	29	31	35	nil	554.16
Sun 3/50	17	24	19	nil	580.66
Total no. of locks obtained (total no. of hops)					437
Time spent in obtaining locks					135.24
Total Processing Time					661.84
HP Utilization					85.34%
Task Imbalance					0.15%

T_t includes the total time spent in obtaining locks which provide the current HPFs for the corresponding locks. Table 3.3 shows the load shared by 16 heterogeneous workstations. The number of hops taken is the number of times a workstation obtained anonymous computing units. The average HPF per hop is indicated as a measure of availability of a workstation. The total number of TSP tree nodes processed by a workstation are indicated in the next column.

Table 3.4 Load Adaptability Test

No. of processors	*Machine architectures*	*Tree size in No. of nodes*	*No. of iterations per Tree Node*	*Average HPF per Hop*	*Load Imbalance (%)*	*HPU (%)*
4	1 IBM RS/6000, 2 Sun Sparcs 1 Sun 3/60	465	1500	1208	1.13	85.85
8	4 IBM RS/6000s, 2 Sun sparcs, 2 Sun 3/60s	1550	2000	5429	0.43	86.40
12	4 IBM RS/6000s, 2 Sun Sparcs 2 Sun 3/60s, 4 Sun 3/50s	2489	2000	2216	0.31	84.65
16	4 IBM RS/6000s, 2 Sun Sparcs, 2 Sun 3/60s, 8 Sun 3/50s	3100	2000	1427	0.15	85.34

When a new task arrives at a heterogeneous cluster for the first time, the source code is transmitted for compilation. The compilation times are also mentioned in Table 3.3. While a task was getting compiled on a node in a cluster, other nodes in that cluster waited for the completion of the compilation process. The last column lists the computation time taken by each workstation. It can be seen that in spite of large variations in speeds and loads of the workstations, a good balance was achieved. An HPU figure is computed as a measure of effectiveness of using the loaded heterogeneous workstations by a program. HPU includes all runtime overheads of the system. A load imbalance figure is computed in terms of the time spent in processing by each workstation. It gives an indication of the load balance achieved by the program using the HPF. Load imbalance is computed as the percentage average deviation from mean processing time.

$$\text{Imbalance} = \frac{\sum_{i=1}^{p} |T_{ci} - T_{\text{c mean}}|}{pT_c \text{ mean}}$$

Table 3.4 shows the results of the test conducted with increasing task sizes on increasing number of nodes. As we increase the task size, the HPU on a loaded heterogeneous system can be maintained at the same high-level with increasing workstation processing power, similar to the trends in the conventional efficiency figure on no load parallel systems. However, our experimentation on dynamic systems shows that it can be extremely difficult to obtain good HPUs on loaded heterogeneous workstations, irrespective of task sizes, if good load balancing is not obtained. The HPF figure can be effectively used in a load balancing scheme that is developed to implement a given application.

Table 3.4 lists four tests conducted on 4–16 workstations at different times. From the average HPF per hop figure as listed in column 5 of Table 3.4, it can be observed that the load in the network varied from test to test. For all the tests, a good HPU was obtained with low imbalance figures. The number of iterations per TSP tree node are also listed in Table 3.4 along with the total number of tree nodes processed as an indication of the overall task size.

It can be concluded from Tables 3.3, 3.4, that ARC enables a parallel program to run in a network of heterogeneous workstations along with other existing sequential programs running in the system. The ARC parallel program has no control over the sequential programs that are started independently on different machines either from respective consoles or through remote logins. Instead, the ARC parallel programs adapt themselves to varying loads and speeds.

3.7.2 The Fault Tolerance Test

The fault tolerance test was conducted by artificially injecting faults by terminating remote processes. The results are summed up in Table 3.5. The *time-out* figure is used for deciding on re-execution in case of a failure. It can be noted that *time-out* does not specify an upper bound on the completion of the task. Whenever a specified time-out is exceeded by a remote execution, a new execution is not tried automatically in the case of a failure. Instead, the failure is reported. However, if the time-out is not exceeded and a failure is detected, a re-execution is tried automatically. An upper bound on the number of re-tries is specified by the *retries* parameter.

Table 3.5 Fault Tolerance Test

	Task: 200 × 200 matrix multiplication divided into four ARC postings			
No. of injected failures	*Time-out specified*	*Retries specified*	*Time-out exceeded by failed units*	*Result of failed ARC postings*
0	1000	0	NA	Successful
1	1000	0	No	Unsuccessful
2	1000	2	No	Successful
0	0	0	NA	Successful
1	0	0	Yes	Unsuccessful
1	0	1	Yes	Unsuccessful

As an example, in Table 3.5, the third test specifies a time-out of 1,000 units and a maximum of two retries. Two artificial failures were injected within the specified time-out, and the ARC posting was successful in its third attempt. By altering the time-out and retries parameters, a desired fault tolerance semantics can be obtained.

3.7.3 The Parallelism No Load Test

This test was conducted by using a double precision matrix multiplication program on a network of homogeneous workstations on no load conditions to compare the overheads of ARC with a widely used traditional COP paradigm, the PVM. The results were compared with the same program implemented by using PVM on no load. Table 3.6 summarizes the speed-up figures obtained for various task sizes. It can be observed that ARC has slightly more overheads than PVM due to an additional message indirection through the ARC kernel. These overheads are due to the fact that ARC provides fault tolerance, load adaptability and automatic handling of heterogeneity not supported by the conventional COP approaches such as PVM.

Table 3.6 No Load Parallelism Test

Task size	*Nodes*	*Speed up for PVM*	*Speed up for ARC*
50 × 50	2	1.81	1.46
100 × 100	2	1.97	1.75
	4	3.72	3.2
	5	4.30	3.8
150 × 150	3	2.96	2.63
	6	5.55	4.75
200 × 200	4	3.84	3.48
	5	4.80	3.48
	8	7.20	6.24

3.7.4 ARC vs Sequential Program

This test was conducted to show the system overhead in the case of a total absence of remote anonymous workstations. In such a case, the program may still execute on the host node dynamically by keeping track of availability through asynchronous intimations as described in Section 3.5.2.3. Table 3.7 shows the results of the no load inlining test conducted on a Sun 3, with a matrix multiplication program with varying problem sizes. Absolute timing figures are given along with percentage slowdown incurred in the case of in-lining. A significant component of the inlining overhead is the lock obtaining time. When a program starts, to obtain a first lock, the system always calls the system coordinator. If a remote anonymous node is not available, a lock is not ensured and the program may inline the code. Any further tries to obtain locks return immediately without success till an asynchronous intimation arrives, thereby indicating the possibility of an available and willing remote node. The asynchronous intimations thus help in reducing the number of unsuccessful calls to the system coordinator. It can be observed from Table 3.7 that an ARC program can continue to execute on the host node without incurring significant slowdowns when the problem sizes are sufficiently large.

Table 3.7 No Load Inlining Test

Task size	*Pure sequential (sec)*	*ARC on same node by inlining (sec)*	*Slowdown (%)*
25 × 25	2.00	2.06	3.00
50 × 50	16.08	16.28	1.25
75 × 75	54.88	55.18	0.55
100 × 100	129.44	130.4	0.75

3.7.5 ARC Calls vs RPC

This test was performed to estimate the overhead of making ARC function calls as compared to remote procedure calls. The tests were carried out on homogeneous machines. A major component in this overhead comes from the dynamic loading of the RIB for execution by the local coordinator on the remote node. A standard matrix multiplication program with varying sizes was executed by using both RPC and ARC function calls. ARC function calls used the same nodes of the network as used by the RPC. ARC showed a 4 to 5 per cent overhead in terms of time as compared to its equivalent RPC call in which the code to be executed was already available at the known server. This additional cost is very nominal as ARC provides the flexibility of choosing a varying number of appropriate nodes for execution at run-time. It can be noted that this overhead is incurred only once in a particular program when it makes multiple ARC calls as in the TSP discussed earlier. The ARC kernel recognizes the availability of an already active RIB code waiting to receive a sub-task eliminating restarting of the RIB again on the remote node.

3.8 Conclusions

The specific issues involved in parallel programming on workstation clusters have been examined in detail. The ARC paradigm has been proposed for parallel programming on loaded heterogeneous cluster of workstations. The ARC model is developed in a two-tiered ARC architecture. The lower level consists of a number of ARC primitives that are built over a distributed ARC kernel. The primitives provide the support for load adaptability, fault tolerance and heterogeneity to ARC-based parallel programs. The ARC model was implemented on a network of heterogeneous workstations and the performance tests showed that ARC is a highly promising approach for parallel programming in practical work environments.

References

1. Agrawal, R. and A.K. Ezzat, "Location-independent Remote Execution in NEST", *IEEE Trans. Software Eng.*, Vol. 13, No. 8, pp. 905–912, August 1987.
2. Amza, C., A. Cox, S. Dwakadas, P. Keleher, H. Lu, R. Rajamony, W. Yu and W. Zwaenepoel, "TreadMarks: Shared

Memory Computing on Networks of Workstations", *Computer*, Vol. 29, No. 2, pp. 18–28, February 1996.

3. Anderson, T.E., D.E. Culler, D.A. Patterson and the NOW Team, "A Case for NOW (Networks of Workstations)", *IEEE Micro*, Vol. 15, No. 1, pp. 54–64, February 1995.
4. Andrews, G.R., "Paradigms for Process Interaction in Distributed Programs", *ACM Computing Surveys*, Vol. 23, No. 1, pp. 49–90, March 1991.
5. Andrews, G.R., R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, "An Overview of the SR Language and Implementation", *ACM Trans. Programming Languages and Systems*, Vol. 10, No. 1, pp. 51–86, January 1988.
6. Bal, H.E., J.G. Steiner, and A.S. Tanenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, Vol. 21, No. 3, pp. 261–322, September 1989.
7. Bal, H.E., M.F. Kaashoek, and A.S. Tanenbaum, "Orca: A Language for Parallel Programming of Distributed Systems," *IEEE Trans. Software Eng.*, Vol. 18, No. 3, pp. 190–205, March 1992.
8. Bal, H.E., R.V. Renesse, and A.S. Tanenbaum, "Implementing Distributed Algorithms Using Remote Procedure Calls," *Proc. AFIPS Conf. National Computing,* Vol. 56, pp. 499–505, Chicago, June 1987.
9. Black, A., N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald," *IEEE Trans. Software Eng.*, Vol. 13, No. 1, pp. 65–76, January 1987.
10. Bershad, B.N., D.T. Ching, E.D. Lazowska, J. Sanislo and M. Schwartz, "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems," *IEEE Trans. Software Eng.*, Vol. 13, No. 8, pp. 880–894, August 1987.
11. Butler, R., and E. Lusk, "User's Guide to the P4 Parallel Programming System," Technical Report ANL-92/17, Argonne National Laboratory, October 1992.

12. Carriero, N., and D. Gelernter, “The S/Net.s Linda Kernel,” *ACM Trans. Computer Systems*, Vol. 4, No. 2, pp. 110–129, May 1986.

13. Carter, J.B., J.K. Bennett, and W. Zwaenepoel, “Implementation and Performance of Munin,” *Proc. 13th ACM SOSP*, pp. 152–164, October 1991.

14. Casas, J., D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole, “MPVM: A Migration Transparent Version of PVM,” Technical Report CSE-95-002, Oregon Graduate Institute, February 1995.

15. Cheriton, D.R., “The V Distributed System,” *Comm. ACM*, Vol. 31, No. 3, pp. 314–333, March 1988.

16. Dahlin, M., R. Wang, T. Anderson, and D. Patterson, “Cooperative File Caching: Using Remote Client Memory to Improve File System Performance,” *Proc. First Conf. Operating Systems Design and Implementation*, November 1994.

17. Eager, D.L. and J. Zahorjan, “Chores: Enhanced Runtime Support for Shared Memory Parallel Computing,” *ACM Trans. Computing Systems*, Vol. 11, No. 1, pp. 1–32, February 1993.

18. Grimshaw, A.S., “Easy-to-Use Object-Oriented Parallel Processing with Mentat,” *Computer*, Vol. 26, No. 5, pp. 39–51, May 1993.

19. Falcone, J.R., “A Programmable Interface Language for Heterogeneous Distributed Systems,” *ACM Trans. Computing Systems*, Vol. 5, No. 4, pp. 330–351, November 1987.

20. Halstead Jr., R.H., “Parallel Computing Using Multi-lisp,” *Parallel Computation and Computers for Artificial Intelligence*, J.S. Kowalik, (ed.), pp. 21–49, Kluwer Academic, 1988.

21. Hansen, B., “Distributed Processes: A Concurrent Programming Construct,. *Comm. ACM*, Vol. 21, No. 11, pp. 934–941, November 1978.

22. Hoare, C.A.R., “Communicating Sequential Processes, *Comm. ACM*, Vol. 21, No. 8, pp. 666–677, August 1978.

23. Hsieh, W.C., P. Wang, and W.E. Weihl, "Computation Migration: Enhancing Locality for Distributed-Memory Parallel Systems," *PPOPP.93*, *ACM SIGPLAN*, pp. 239–248, July 1993.

24. Janakiram, D., T.H. Sreenivas and G. Subramanyam, "Parallel Simulated Annealing Algorithms, *J. Parallel and Distributed Computing*, Vol. 37, pp. 207–212, 1996.

25. Joshi, R.K. and D. Janakiram, "Parset: A Language Construct for System-independent Parallel Programming on Distributed Systems," *Microprocessing and Microprogramming*, pp. 245–259, June 1995.

26. Joshi, R.K. and D. Janakiram, "Object-based Subcontracting for Parallel Programming on Loosely-Coupled Distributed Systems," J. Programming Languages, Vol. 4, pp.169–183, 1996.

27. Kale, L.V, and S. Krishnan, "CHARM++: A Portable Concurrent Object–Oriented System based on C++," OOPSLA.93, ACM SIGPLAN Notices, pp. 91-108, October 1993.

28. Minnich, R.G., "Mether: A Memory System for Network Multiprocessors," PhD thesis, Computer and Information Sciences, University of Pennsylvania, 1991.

29. The MPI Forum, "MPI: A Message Pssing Interface," *Proc. Supercomputing*, Vol. 93, pp. 878–883, 1993.

30. Mullender, S.J., G. Rossum, A.S. Tanenbaum, R. Renesse, and H. Staveren, "Amoeba: A Distributed Operating System for the 1990's," *Computer*, Vol. 23 No. 5, pp. 44–53, May 1990.

31. Nelson, B.J., "Remote Procedure Calls," *ACM Trans. Computing Systems*, Vol. 2, No. 1, pp. 39–59, February 1984.

32. Pekergin, M.F., "Parallel Computing Optimization in the Apollo Domain Network," *IEEE Trans. Software Eng.,* Vol. 18, No. 4, pp. 296–303, April 1992.

33. Satyanarayanan, M., J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel, and D. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Trans. Computers,* Vol. 39, No. 4, pp. 447–459, April 1990.

34. Satyanarayanan, M. and E. Siegel, "Parallel Communication in a Large Distributed Environment," *IEEE Trans. Computers,* Vol. 39, No. 3, pp. 328–348, March 1990.

35. Scott, M.L., "Language Support for Loosely Coupled Distributed Programs," IEEE Trans. Software Eng., Vol 13, No. 1, pp. 88–103, January 1987.

36. Smith, J.M., "A Survey of Process Migration Mechanisms," *ACM SIGOPS*, Vol. 22, No. 3, pp. 28–40, July 1988.

37. Stamos, J.W., and D.K. Gifford, "Implementing Remote Evaluation," *IEEE Trans. Software Eng.*, Vol. 16, No. 7, pp. 710–722, July 1990.

38. Stevents, W.R., "*Unix Network Programming*", Englewood Cliffs, N.J., Prentice Hall, 1990.

39. Stumm, M. and S. Zhou, "Algorithms Implementing Distributed Shared Memory," *Computer*, Vol. 23, No. 5, pp. 54–64, May 1990.

40. Sunderam, V.S., "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience,* Vol. 2, No. 4, pp. 315–339, December 1990.

41. Tandiary, F., S.C. Kothari, A. Dixit, and W. Anderson, "Batrun: Utilizing Idle Workstation for Large-Scale Computing," *IEEE Parallel and Distributed Technology,* pp. 41–49, 1996.

42. Wei, Y., A.D. Stoyenko, and G.S. Goldszmidt, "The Design of a Stub Generator for Heterogeneous RPC Systems," *J. Parallel and Distributed Computing*, Vol. 11, No. 3, pp. 188–197, March 1991.

Chapter 4

Integrating Task Parallelism with Data Parallelism*

4.1 Introduction and Motivation

Data parallelism refers to the simultaneous execution of the same instruction stream on different data elements. Several programming platforms target the exploitation of data parallelism [1, 2]. Control parallelism refers to the simultaneous execution of different instruction streams [2]. This is also referred to as task parallelism or functional parallelism [2]. Some of the tasks that constitute the problem may have to honour precedence relationships amongst themselves. The control parallelism with precedence constraints can be expressed as a task graph wherein nodes represent tasks and directed edges represent their precedences. It is the parallel execution of distinct computational phases that exploit a problem's control parallelism [3]. This kind of parallelism is important for various reasons. Some of these reasons are:

- *Multi-disciplinary applications:* There is an increased interest in parallel multi-disciplinary applications wherein different modules represent different scientific disciplines and may be implemented for parallel computation [4]. As an example, the airshed model is a Grand Challenge Application that characterizes the formation of air pollution as the interaction

*K. J. Binu, D. Janakiram.

between wind and reactions among various chemical species [5].

- *Complex simulations:* Most of the complex simulations developed by scientists and engineers have potential task and data parallelism [6]. A data parallel platform would not be able to exploit the potential control parallelism in them.
- *Software engineering:* Independent of issues relating to parallel computing, treating separate programs as independent tasks may achieve benefits of modularity [7].
- *Real-time requirements:* Real-time applications are characterized by their strict latency time and throughput requirements. Task parallelism lets the programmer explicitly partition resources among the application modules to meet such requirements [4].
- *Performance:* Task parallelism allows the programmer to enhance locality and hence performance by executing different components of a problem concurrently on disjoint sets of nodes. It also allows the programmer to specify computation schedules that could not be discovered by a compiler [7].
- *Problem characteristics:* Many problems can benefit from a mixed approach, for instance, with a task parallel coordination layer integrating multiple data parallel computations. Some problems admit both data and task parallel solutions, with the better solution depending upon machine characteristics or personal taste [7].

Very often, task and data parallelism are complementary rather than competing programming models. Many problems exhibit a certain amount of both data parallelism and control parallelism. Hence, it is desirable for a parallel program to exploit both data and task parallelism inherent in a problem. A parallel programming environment should provide adequate support for both data and task parallelism to make this possible [8].

4.2 A Model for Integrating Task Parallelism into Data Parallel Programming Platforms

4.2.1 Expectations from an Integrated Platform

The expectations from a high-level parallel programming platform stem from the nature of applications which could utilize the platform. The requirements come from the desired expressibility of the application, possible transparency in programming, exploitation of parallelism, and achievable performance optimizations for the application. There factors are detailed below.

- *Expressibility:* In order to exploit parallelism in an application, the program must express potential parallel execution units. The precedence relationships among them must also be expressed in the program. An elegant expressibility scheme should reflect the task parallel units, data parallel units and precedence dependence among the tasks in the program. This would ease programming, improve readability and enhance maintainability of the code. However, the expressibility that can be provided is influenced by the nature and organization of the underlying run-time support and the native language into which it is converted.
- *Transparency:* It is desirable to relieve the programmer from details relating to underlying network programming. This results in the programmer concentrating on his application domain itself. With network programming details coded in the application, a major portion of the program will be unrelated to the application. Consequently, such programs suffer from readability and hence maintainability problems.
- *Performance:* System level optimizations by the parallel programming platform can improve the performance of applications. In addition, the system can achieve load balancing for the application, thereby further enhancing performance. The run-time scheduling decisions by the system, both the time of scheduling and the node to be scheduled, are the other factors that can improve performance.

Other desirable properties of the system include fault resilience, fault tolerance, accounting for heterogeneity in machine architecture and operating system, and portability of application.

4.2.2 Programming Model

This model permits a block-structured specification of the parallel program with arbitrary nesting of task and data parallel modules. This is one of the most important qualities of an integrated model [8]. A block could be a high-level characterization of a data parallel module in accordance with the principles of the underlying data parallel platform. In the model, the system takes the responsibility of intimating the user process when an event of its interest occurs. Events of interest signify the completion of one or more tasks which meet the pre-conditions for another task that is waiting to be executed. This takes care of the probability factor in the order of completion of tasks that constitute the program. The model provides for templates through which events of interest could be registered with the system.

The model aims at a parallel programming platform that permits expressibility for task and data parallelism so that both can be exploited. In view of a large number of existing data parallel programming platforms for NOWs, it would be useful to formulate the problem as integrating task parallelism into existing data parallel platforms. This poses some challenges. The data parallel model of computation of these platforms could be different. Consequently, the program structure favoured by these platforms would be different. The system services provided by them, viz. migration, result collection, etc., could also be different. Hence, at the programming level, the model restricts itself to expressing tasks and their precedence relationships. This makes sure that an integration of the model to an existing data parallel platform does not contradict its existing goals and the rules by which it handles data parallel computation. Also, it inflicts only minimal disturbance on the existing system.

Another issue which crops up during the integration pertains to data parallel sub-division of divisible tasks. The underlying data parallel model could be sub-dividing a data parallel task into sub-tasks. These sub-tasks could be migrated to different nodes over the network. The underlying data parallel platform could be considering sub-tasks as separate entities from the time it is spawned to its result collection. The platform would have had no purpose to identify a task as a collection of sub-tasks that constitute it. However, the completion of a task could be a significant event in the proposed integrated model since the system has to intimate the user process when an event of its interest occurs. The completion of a task depends upon the completion of the sub-tasks into which it is split by the underlying data parallel model. Hence, it becomes necessary to integrate to the existing system, the notion of a task as a collection of sub-tasks.

The program expressibility of the model reflects the task parallel blocks and precedence relationships in the task graph. Two constructs, viz. *Task begin* and *Task end*, are introduced to demarcate the blocks in the block-structured code. Another construct, viz. *OnFinish*, is provided to specify the pre-conditions of the tasks. The syntax and semantics of these constructs are described below.

```
Task_begin(char *TaskName) OnFinish(char
*WaitforTask, ...)
```

This marks the beginning of a block. Each block signifies a task. The task name of the task it signifies is the only argument to the construct. If the task has precedence relationships to be met, the *Task_begin* has to be immediately followed by another construct, viz. *OnFinish* with the names of the tasks it has to wait for as its arguments. If *OnFinish* is not furnished, the parser will presume that there are no pre-conditions to the task. *OnFinish* could have a variable length of arguments.

```
    Task_end(char *TaskName)
```

This marks the end of a block. The only argument is the name of the task that the block signifies.

4.2.3 Program Structure and Translation of a Task Graph

A sample task graph and its block-structured code are shown to illustrate the expressibility provided by the model. Also, it illustrates the translation of a given task graph into the program structure favoured by the model. The task graph given in Fig. 4.1 is used for the same.

```
Task_begin(Task1)
    ...
    Task_begin(Task3) OnFinish(Task1)
    ...
    Task_end(Task3)
  Task_end(Task1)
Task_begin(Task2)
    ...
    Task_begin(Task4) OnFinish(Task1, Task2)
    ...
    Task_end(Task4)
    Task_begin(Task5) OnFinish(Task2)
    ...
      Task_begin(Task6) OnFinish(Task3, Task4,
      Task5)
        ...
        Task_end(Task6)
      Task_end(Task5)
Task_end(Task2)
```

Pseudo Code 1: The block structuring corresponding to the task graph in Fig. 4.1.

The outline program given in Fig. 4.1 shows the expressibility of a task graph in the model. At the first level, *Task1* and *Task2* could be executed in a control parallel fashion at the beginning of the run itself. This is evident from their *Task begin* constructs, since it is not followed by the construct *OnFinish*. A task which

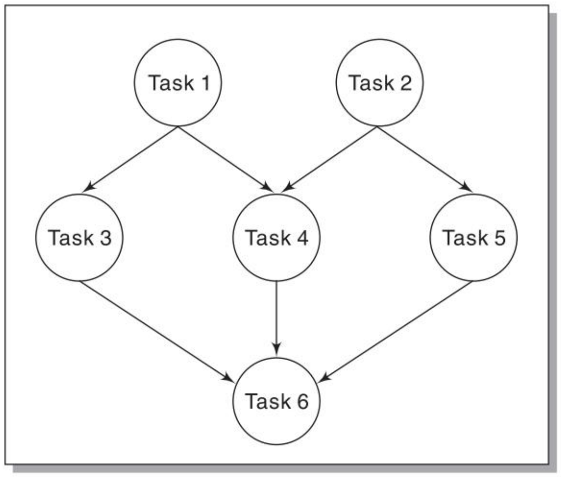


FIG. 4.1

A Sample Task Graph with Precedence Relationships

has to wait for another task to finish is written inside its predecessor task's block. Also, the *Task begin* construct of such tasks would be immediately followed by the construct *OnFinish* which specifies the pre-conditions. In the case where a task has to wait for more than one task, it can be placed inside one of those blocks which represents a predecessor task. But, its *OnFinish* should carry the name of all its immediate predecessors. This could be seen from the *OnFinish* directives of *task4* and *task6*.

Typically, data parallel platforms split divisible tasks into sub-tasks. Hence the model provides for a call, viz. *task()*, to register a task as a collection of sub-tasks. The syntax and semantics of the call are described in Section 4.3.3.

4.2.4 Separation of System's and Programmer's Concerns

The system is organized in such a way that the user process registers events of interest (completion of one or more tasks) which are required in order to meet the precedence constraints and the system, in turn, signals the user process on occurrence of any registered event. This relieves the programmer of the need to design applications for non-determinism in the order of completion of tasks.

With the program translation capabilities that are provided, the model relieves the programmer of the need for using network-related code. Another responsibility lying with the translator is the need to contend with the conflicting interests of expressibility and performance. Arguments from the expressibility point of view favour a control structure that best reflects the precedence relationships in the code. This, in turn, would create an argument for the structuring of programs such that the pieces of code that are executed at the completion of its predecessor task appear as a program segment inside the program segment of (one of) its predecessor task itself. Such a structure suitably describes the task graph and contributes to the elegance of the code. However, it does not support non-determinism in the order of completion of tasks according to the flow of control permitted by traditional languages. Hence, in order to allay the differences between these demands, the program translator allows a program expressibility scheme, which satisfies the expectations from the expressibility point of view. This, in turn, is parsed and translated into a control structure that well answers the concerns of non-determinism.

The program translation provided with the current model permits a control structure similar to the block-structured code that programmers are familiar with. At the same time, it reflects the task graph and precedence relationships of the application. The parser and sample translated code are described in section 4.4.1.

4.3 Integration of the Model into ARC

4.3.1 ARC Model of Computation

In the ARC paradigm [9], a parallel program for NOW is written as a collection of several loosely coupled blocks called RIBs within a single program entity. An RIB is a code fragment that can be migrated into a convenient anonymous remote node at runtime for execution. RIBs do not involve any mechanism of process creation or inter-task communication at the programming language level. The nodes at which RIBs need to be executed remain anonymous to the program. The ARC runtime system decides the nodes for RIB execution. At a given time, multiple programs could be generating RIBs and multiple anonymous remote participants could be joining or leaving the ARC system. ARC addresses heterogeneity in the architecture and operating system, fault tolerance, load balancing with other loads co-existing, and resilience to the changing availability of nodes. However, ARC targets data parallel applications. The control parallelism along with its precedence relationships cannot be expressed in ARC.

In order to achieve load balancing, the ARC model provides a system service which can be used by the user program to get the current availability and load of machines. The user program can use this information, along with the machine handles returned by the call, to migrate his RIBs to least loaded machines. For data parallel modules, the load ratio on machines could be used to decompose the domain of computation to achieve load balancing. Subsequently, each grain of computation can be migrated to the corresponding machines, along with the relevant initial data. The ARC model provides two calls for functionalities related to result collection. By availing of these services, the user program can get the status of a submitted task as well as collect the results when they are ready. The syntax and semantics of the calls supported by ARC are given below.

```
LFmessage get_load_factor(int numberOfMachines
Needed)
```

```
    This call is used to obtain information
about various machines available in the system
and their loads. The parameter to this call is
the number of machines required by the program.
The return value is a structure which gives the
number of machines actually available, their
load information and machine indices, and is
used by the underlying system to identify the
machine.
```

`ARC_function_call(char *funct_name, int timeout, int retries, char * arg_data, int arg_size, int result_size, int machine_index, int tag)`

This call is used to submit a task. The first argument specifies the function that specifies the task. The second argument is the time-out period. The third argument is the number of retries that should be made. The fourth argument is the raw data that represents the arguments to the task. The fifth argument is the size of the result. The next argument is the index of the machine, obtained by using *get_load_factor* call. The last argument is the tag to identify the particular task submitted. The value of the tag is returned by the call.

`char *obtain_results(int tag);`

This call is to collect the results of a task. The only argument is the unique *id* of the task. The call blocks till the results are available.

`int peek_results(int tag);`

This is a non-blocking call for a program to check if the results are available. The only argument is a tag for the piece of computation of which the result is requested. The return value is 1 if the results are available, otherwise it is -1. When the results are available, it can be collected by *obtain_results* call.

4.3.2 Outline of ARC Runtime Support

This sub-section outlines the runtime support of ARC [10]. The support consists of a daemon running on each machine present in the pool of machines which cooperate for parallel computation. This daemon is termed 'local coordinator' (*lc*). It is the responsibility of the *lc* to coordinate the processes initiated on the machine on which it runs. Any process which intends to make use of the system services registers itself with the *lc* that runs on its machine. This is done by the call *initialize ARC*. Complementing the call is *close ARC* which deregisters the user process from the system. The syntax and semantics of the calls are given below.

```
    void initialize_ARC(void);
```

`It takes no arguments and returns no values. It registers the user program with the runtime system. In response to this call, the lc allots an exclusive communication channel between the lc and the user program for subsequent communication.`

```
    void close_ARC(void);
```

`It takes no arguments and returns no values. It de-registers the user program from the runtime system. The system updates its tables accordingly.`

One of the machines in the pool is selected to run a daemon which co-ordinates the *lc*s of all the machines which participate in the pool. This daemon is termed 'system co-ordinator' (*sc*). The *sc* keeps track of the *lc*s in the pool and hence the machines participating in the pool. It also facilitates communication between individual *lc*s. The *lc* and *sc* communicate by virtue of predefined messages (through a dedicated channel). Similarly, the user process communicates with the *lc* through predefined messages. In a typical session, a message session sequence would be initiated by the user process by sending a message to its *lc*. The *lc*, in turn, sends the appropriate message to the *sc*. The *sc* may either reply to this message or send a message to another *lc* if required. In the first case, the *lc,* on receiving the message, would generate and send a message to the user process which initiated the sequence. In the latter case, the *lc*, which receives the message from the *sc,* would send a reply on completion of the responsibility. This message would be passed to the *lc* which initiated the sequence. The *lc* would generate and send a message to the actual user process which initiated the message sequence.

4.3.3 The Integrated Platform

The integration of the model into the ARC framework includes, provision for additional function calls and modifications to the existing runtime support. The crux of such modifications is presented in this section.

The additional calls supported include a call to register a task as a collection of sub-tasks, calls to initialize and close the system, and a call to register events of interest with the system. The ARC model permits the division of the number of sub-tasks at runtime. Different tasks could also be divided into different numbers of grains. For these reasons, the call to register a task as a collection of its sub-tasks, viz. *task()*, provides for a variable length argument. The responsibilities of the calls to initialize and close the system, viz. *TaskInit()* and *TaskClose()*, are trivial. The call to register the event, viz. *RegisterEvent()*, is inserted by the parser during program translation. The syntax and semantics of the calls are given below.

```
int task(char *TaskName, int SubTaskId,
...)
```

The call registers a task as a collection of sub-tasks. The first argument is the task name itself. The second argument is an integer which denotes a sub-task. If the task is not sub-divided, the *task_no* itself is to be furnished here. If the task is sub-divided, there could be more arguments, each of which represents a sub-task. The number of arguments depends upon the number of sub-tasks that constitute the task. The return value is either SUCCESS or ERRNO corresponding to the error.

```
int RegisterEvent(int EventId, char* TaskNames,
...)
```

The call registers an event of interest with the system. The first argument is the event identifier. Following it is a variable length argument list of task names constituting the event. The return value is either SUCCESS of ERRNO corresponding to the error.

The modifications to the runtime support of ARC could be summarized as additional table management, additional message protocols between daemons, and minimal changes to the system's response to some existing messages. Additional table management includes mapping tasks to their sub-tasks, storing the events of interest for a process and the status of finished tasks. Additional message protocols are those which access and modify these tables. There has been an inevitable change in the existing system. This was to make the system generate a message to a user process when an event of its interest occurs. In ARC, a task can be considered to have finished its execution only when all its sub-tasks finish their execution. The local *lc* gets a message from remote *lc*s when the sub-tasks given to them finish their execution. The ARC *lc* stores the result and waits for the user process to ask for it. In the integrated system, the *lc* has an additional responsibility when a sub-task returns. It checks if the sub-task that has finished is the last sub-task of the task. If the task can be considered to have finished its execution, it would check the status of finished tasks and events of interest to find out if any event of interest to the user process has occurred with the completion of the task. If an event of interest has occurred with the completion of the task, it initiates a message sequence to intimate the event to the corresponding user process. Additional messages are defined in order to intimate the event of interest to the corresponding user process. These messages are discussed in the Appendix.

4.3.4 A Sample Block in the Integrated Platform

The program structure of the integrated platform would be the same as the one shown with the model. It has been mentioned earlier that the code inside a block would be meant for the platform to which the model is integrated. A sample block is given to provide a comprehensive view of the integrated platform. The semantics of the calls have been discussed in Section 4.3.1.

```
Task_begin(TaskN) OnFinish(TaskM)
// Collect the results of an earlier task
for subsequent processing.
// The earlier task was data parallelized
into two parts with
```

```
// TaskMTag1 and TaskMTag2 representing sub-
tasks.
ObtainResults(TaskMTag1);
ObtainResults(TaskMTag2);
// GetLoadFactor returns a structure which
gives details of the available machines,
their loads, processing powers etc.
// The only argument specifies the maximum
number of machines sought for.
MachineAvailability = GetLoadFactor(3);
if (MachineAvailability.Count == 3)
{
// Data parallelize into three with the
division based on the
// load and processing power of three
machines returned.
ARC_function_call(TaskN, ..., TaskNTag1);
ARC_function_call(TaskN, ..., TaskNTag2);
ARC_function_call(TaskN, ..., TaskNTag3);
task(TaskN, TaskNTag1, TaskNTag2, TaskNTag3);
}
if (MachineAvailability.Count == 2)
{
// Data parallelize into two with the
division based on the
// load and processing power of two machines
returned
  ARC_function_call(TaskN, ..., TaskNTag1);
  ARC_function_call(TaskN, ..., TaskNTag2);
  task(TaskN, TaskNTag1, TaskNTag2);
}
if (MachineAvailability.Count == 1)
{
```

```
// Cannot be data parallelized due to non-
availability of nodes. Hence run as a
sequential program.
  ARC_function_call(TaskN, ..., TaskNTag1);
  task(TaskN, TaskNTag1);
}
Task_end(TaskN)
Pseudo Code: A typical block in the
integrated platform.
```

The code shows how ARC decides the number of sizes of grains of computation at runtime after collecting the load information. Also, note that the arguments to the *task()* call reflect the actual division employed. The block is marked by *Task begin* and *Task end*. *OnFinish* specifies the pre-condition of *TaskN* as the completion of *TaskM*.

4.4 Design and Implementation

The parser for program translation, the *lc* daemon for user process coordination, the *sc* daemon for coordination of the pool of workstations and functional library support to avail system services, are the constituent elements of the system.

4.4.1 Parser

The parser translates the program submitted by the user into the final runnable program. This involves insertion of appropriate network-related code, translation of the pseudo-control structure provided by the model to one which is supported by the native language, construction of events of interest to the user process, and transparent insertion of some system service calls.

In the first scan of the user-submitted program, the parser constructs the precedence graph. It marks the tasks which do not have any precedence relationship to be met. It finds the events of interest to the user process. The calls to register these events with the system are inserted in the code. A header file is generated to define the event names and the file is included in the user program file. This permits event names themselves to be used as cases in the final flat switch-case structure. The tasks which do not have to meet any precedence relationships are extracted and placed before the flat switch-case structure since they can be executed without waiting for any events.

In the second scan, the parser constructs an infinite loop which waits on a socket to read event interrupts. A switchcase structure is placed inside the loop with event names as cases. The body of a case is the code fragment of the task which waits for the event. This control structure is a flat one, unlike the control structure permitted by the model. The termination would become part of the case which corresponds to the final task.

The prototypic version of the parser developed can convert the code only into one native language, viz. C. It is relatively straightforward to extend the scope of the parser to cater to other native languages.

Given below is a sample translated code. The code is obtained by translation of Pseudo Code 1. The transformations by the parser could be seen by mapping the sample translated code with Pseudo Code 1.

```
/* Sample Defines file */
#define EVENT_Task1 1
#define EVENT_Task2 2
#define EVENT_Task1_Task2 3
#define EVENT_Task3_Task4_Task5 4
/* Register Events */
RegisterEvent(EVENT_Task1, "Task1")
RegisterEvent(EVENT_Task2, "Task2");
RegisterEvent(EVENT_Task1_Task2, "Task1",
"Task2");
RegisterEvent(EVENT_Task3_Task4_Task5,
"Task3", "Task4", "Task5");
/* Contents of the Block for Task1 */
...
```

```
    /* Contents of the Block for Task2 */
    ...
    while(1)
    {
    Event = WaitForEvent();
    switch (Event)
      {
      case EVENT_Task1 :
    /* Contents of the Block for Task3 */
    ...
    break;
    case EVENT_Task2 :
    /* Contents of the Block for Task5 */
    ...
    break;
    case EVENT_Task1_Task2 :
    /* Contents of the Block for Task4 */
    ...
    break;
    case EVENT_Task3_Task4_Task5 :
    /* Contents of the Block for Task6 */
    ...
    exit();
    break;
      }
    }
Pseudo-code : A translated code for Pseudo Code
1
```

4.4.2 Local Co-ordinator (*lc*) Daemon

Each workstation which enrols in the pool of machines for parallel computation runs a daemon, viz. *localdaemon(lc)*. The user processes communicate and interact with the system through the *lc*. Servicing requests from the user processes, linking the node on which they are run to the system, and book-keeping for the user process running on its machine, are the responsibilities of the *lc*.

Figure 4.2 gives the Finite State Machine (FSM) of the *lc*. In the **INIT** state, *lc* initializes its data structures and cleans up the auxiliary system files left behind by an earlier *lc* process. After this, it establishes a TCP socket connection with the *sc* and registers itself with the *sc*. In the **LISTEN** state, the *lc* checks for messages from the *sc* and waits for user programs to register with the system. On a message from the *sc*, it transits to the **SC Msg RECVD** state where it services the message. On a message from a user process, it transits to **UP Msg RECVD** where it services the request from the user process.

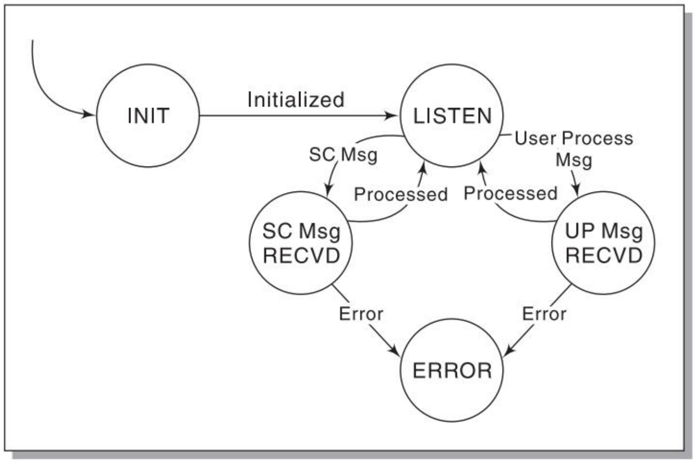


FIG. 4.2

FSM of LC

The initial communication between a process and the *lc* is through a known common channel. This is for a user process to get itself registered with the *lc*. Once it is registered with the *lc*, it is given an exclusive communication channel though which subsequent communication is effected. When the *lc* is initiated, the communication with the *sc* is effected by giving the address of the machine on which the *sc* is run as a command line argument.

The tables maintained by the *lc* could be distinguished as those required to support task parallelism and others. This separation eases a clean integration of task parallelism into the existing systems.

In ARC, the *lc* maintained three tables, viz. the Program and Task Table (PTT), Recovery Information Table (RIT) and Results List Table (RLT). PTT maintains the mapping between the process ids of the processes on the machine and the socket descriptors of the sockets that connect the processes to the *lc*. It also demarcates processes as user processes initiated on the node and the tasks submitted by user processes on other nodes. RIT maintains information that is relevant for recovery in the event of a failure. RLT stores the results of the tasks submitted by user processes on the machine as and when they are available. This is stored until they are claimed by the user process.

It can be seen that the system does not keep track of the sub-tasks which constitute a task. The significance of keeping track of sub-tasks that constitute a task has already been discussed. Hence, a new table, viz. Task Table (TT), is maintained which maps tasks to the sub-tasks into which it is divided. This mapping is done on a per process basis. The table is updated when a task is sub-divided. The programming language interface for updating of the table has already been discussed.

In the integrated system, the *lc* keeps track of the events of interest to the user processes. It maintains a table, viz. Event Table (ET), for this purpose. It stores the pre-declared events of interest on a per process basis. The interface for updating of the table is inserted by the parser by interpreting the events of interest from the user code.

Other than the additions of the above-mentioned tables, their programming language level interfaces and the message sequences that they initiate, there are some modifications to the existing *lc*. This is to enable the *lc* to inform the user process when an event of interest occurs. A task finishes its execution when the last sub-task of the task finishes. Hence, when the *lc* gets an intimation of the completion of a sub-task from the *sc*, it checks if the task has finished its execution with the sub-task that has returned. If the task is over, the *lc* checks to see if it triggers any event of interest. If so, the *lc* initiates a message to the user process intimating the occurence of the event. TT and ET are the tables consulted by the *lc* in order to accomplish this job. In cases where completion of a sub-task results in more than one event of interest, they are intimated with separate messages one after the other. Given below are the structures of ET and TT.

```
struct EventTable
{
int EventIdentifier; // Event Name
char ** WaitForTaskNames; // Task Names for
this Event
BOOLEAN* TasksOver; // Task Completed Array
int NumberofWaitForTask; // Number of Task
for the Event
}
struct TaskTable
{
  char TaskName[TASKNAME_LENGTH];
                                  // Name of
  the Task
  int * SubTaskIdentifier; // Sub task
  indentifiers
  BOOLEAN * SubTaskOver; // Sub task
  completion status
  int NumberofSubTask; // Number of sub
  tasks
}
```

4.4.3 System Coordinator (*sc*) Daemon

The *sc* coordinates the set of *lc*s that constitutes the system. The *sc* maintains the global information of the system. Also, for small sessions, the *sc* routes the messages between *lc*s so that the overhead for frequent connection establishment and closing is minimized. The *sc* is connected to *lc*s through TCP sockets. The message structure includes a field to indicate the destination address in order to facilitate this. The *sc* that is employed by the ARC is a fairly thin deamon with minimal information stored, and hence it scales up quite well with respect to the number of *lc*s that participate in the pool.

As shown in Fig. 4.3, in the state **INIT**, *sc* cleans up the current directory for any auxiliary system files left behind by an earlier *sc* process. It then initializes its structures. In the **LISTEN** state, the *sc* polls for connection requests from *lcs* and registers them with the system, establishing a TCP socket between itself and the *lc*. It then listens for messages from the *lcs* on these exclusive channels and services the requests. The *sc* remains in this state throughout its lifetime or until it encounters an error.

Our implementation for the integration of task parallelism into ARC does not disturb the existing *sc*.

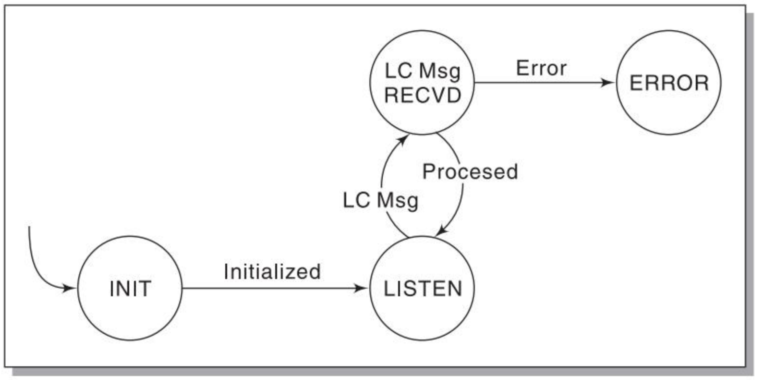


FIG. 4.3

FSM of SC

4.5 Applications

Applications with coarse grain control parallelism or coarse grain data parallelism or both are the target of our platform. We have already discussed some broad classes of applications that could potentially benefit from the platform. In addition, here we will discuss some specific applications.

Many applications in signal processing have potential coarse grain control and data parallelism. A good proportion of signal processing applications do some form of signal transformation. These transformations are typically followed by a filtering module, which filters the transformed signal. The inverse of the transformation could follow the filtering stage. As an example, an application could consist of a Fourier transform followed by some filter module that is dependent on the purpose of the application which is followed by the inverse Fourier transform. Fourier transform and its inverse are amenable to data parallel computation. Similar is the case with many other signal transforms. The exploitation of parallelism in signal processing becomes even more important in real-time signal processing.

Here, we discuss the exploitation of parallelism in one such application, viz. the speaker verification problem. The speaker verification program starts with a sample of the time domain signal. A solution to the problem starts with a linear predictive analysis [11] of the input time domain signal to yield Linear Predictive (LP) coefficients. From the set of LP coefficients, LP Cepstrums, which are features of the input signal, are obtained. There are different methods to obtain evidence for verification from the LP Cepstrums: the Gaussian Mixture Model (GMM) method [12], neural network methods, etc. are some examples. Some methods prove better than the rest according to the nature of the input set. Different methods could be applied control parallely on the same set of LP Cepstrums. The Constraint Satisfaction Model (CSM) [13] combines evidence obtained from each of these models. Each of the methods, in turn, could exploit the data parallelism in the problem.

Given below is a sample pseudo-code for the application on the integrated platform, the task graph of which is given in Fig. 4.4.

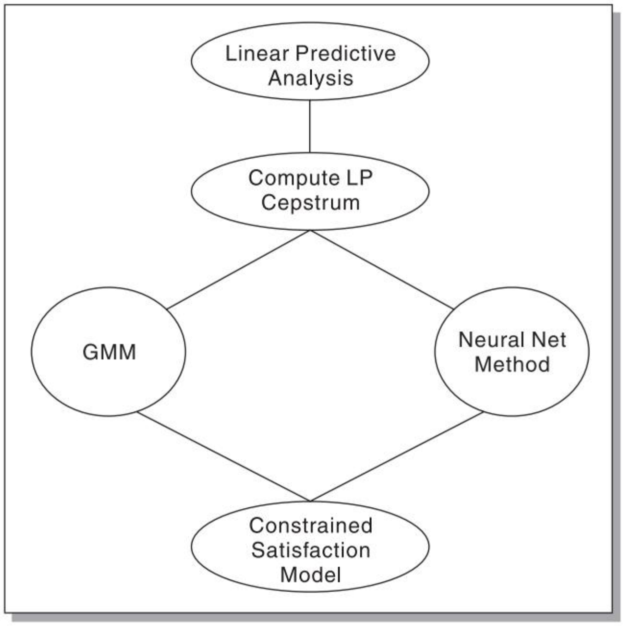


FIG. 4.4

Task Graph of the Application

```
LPAnalyse();
ComputeLPCepstrums();
// GMM and NeuralNet blocks are task
parallelized
Task_begin(GMM) // Start of GMM ARC block
    // migrate to four lightly loaded nodes
    GetLoadFactor(4);
// Data parallelized
      ARC_function_call(GMM,...,GMMTag1);
```

```
          ARC_function_call(GMM,...,GMMTag2);
          ARC_function_call(GMM,...,GMMTag3);
          ARC_function_call(GMM,...,GMMTag4);
      Task_end(GMM)
      Task_begin(NeuralNet) // Start of NeuralNet
      ARC block
      // migrate to three lightly loaded nodes
      GetLoadFactor(3);
                  // Data parallelized
    ARC_function_call(NeuralNet,...,NeuralNet
    Tag1);
    ARC_function_call(NeuralNet,...,NeuralNet
    Tag2);
    ARC_function_call(NeuralNet,...,NeuralNet
    Tag3);
        // Wait for GMM and NeuralNet to complete
        Task_begin(ConstriantStatisfactionModel)
        OnFinish(GMM, NeuralNet)
        ObtainResult(GMMTag1);
        ObtainResult(GMMTag2);
        ObtainResult(GMMTag3);
        ObtainResult(GMMTag4);
        ObtainResult(NeuralTag1);
        ObtainResult(NeuralTag2);
        ObtainResult(NeuralTag3);
    CalculateConstriantStatisfactionModel();
    Task_end(ConstriantStatisfactionModel)
    Task_end(NeuralNet)
```

4.6 Performance Analysis

This section presents performance-related aspects of the work. The test bed for the experiments consists of a heterogeneous collection of interconnected workstations with other loads co-existing. The problem that is considered has control as well as data parallelism. The task graph of the problem is given in Fig. 4.5.

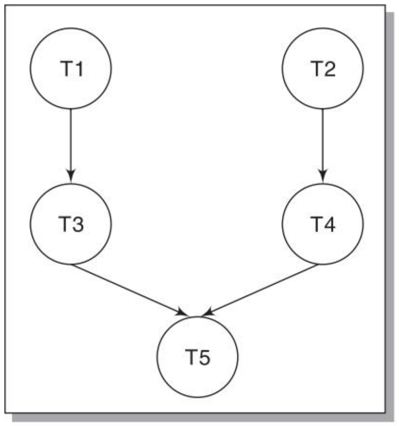


FIG. 4.5

Task Graph of the Application

As the program starts its run, the two tasks, viz. T1 and T2, can start their execution. These two tasks perform matrix multiplication on two different sets of large matrices. Task T3 is a user-defined function to operate on the resultant matrix of T1. Similarly T4 is a user-defined function to operate on the resultant matrix of T2. T3 and T4 can start execution only after the respective completion of T1 and T2. The average completion time registered by T1 and T2 on a representative single machine falls between 20 and 25 minutes. These measurements occur at relatively lightly loaded CPU conditions, though there could be spurts of loads that occur during the period of the run. The computational requirements of T3 and T4 are dependent upon the values of the input itself. This would cause another probabilistic factor in the completion time of these tasks. Consequently, T3 and T4 register an average completion time in the range of 8–20 minutes under the same conditions. T5 is executed when T3 and T4 complete their execution. In the problem considered, T5 is a thin task with its only responsibility being collecting the results of its two predecessor tasks. The task completion time of T5 is insignificant and hence not considered for the analysis.

The control parallelism in the problem is exploited by independent execution of two control parallel arms in the task graph. The data parallelism in the problem is through data parallel execution of each of the tasks T1, T2, T3 and T4.

In the first experiment, tasks T1 to T4 are executed in a data parallel fashion on three nodes each. The run is repeated five times and the time of completion of each of the tasks is observed. During data parallel division, the existing load conditions are taken. The grain size of individual data parallel sub-tasks is determined by using these load snapshots.

The time of completion of a task is shown in Table 4.1 as its critical path. The terminology is adopted because a task is said to have completed when the last among its sub-tasks is completed. Along with the time of completion of T1 to T4 is given the larger value of the time of completion of the first and second arm of the task graph. The last column gives the difference in the time of completion of the two arms.

Table 4.1 Some Sample Scenarios (Time in Minutes)

No.	*Crit_T1*	*Crit_T2*	*Crit_T3*	*Crit_T4*	*CritPath*	*Diff_Crit_Path*
1	7.9	9.1	5.3	4.7	13.8	−0.6
2	8.1	10.8	4.8	4.7	15.5	−2.6
3	11	8	7	5	18	+5.0
4	8	10.5	7.2	4.9	15.4	−0.2
5	11.2	8	4.8	7.1	16	+0.9

The observations of interest from the experiments can be summarized as follows:

- In spite of a task division policy based on runtime load conditions, the completion time of tasks could vary considerably. It can be seen from Table 4.1 that T1 registers a high of 11.2 in the fifth observation against a low of 7.9 in the first observation, yielding a difference of 3.3 minutes, which is 41 per cent of the lower value.

- Any of the control parallel arms could finish before the other and the difference in their completion time could be substantial. A negative value of difference in critical path signifies the first arm finishing before the second and vice versa.
- The difference in critical paths of the arms could have cumulating or compensating effects from the individual completion times of the tasks in the arms. Observation 3 show the cumulating effect whereas observations 4 and 5 show the compensating effect.
- The values presented are taken without inducing any artificial loads. Under heavy load fluctuations or with artificially altering load, the probabilistic values would fluctuate even more.

The earlier experiment has brought out the probabilistic factors in the time of completion of tasks. The next set of values presents the effect of pre-supposing the order of completion of tasks to schedule the subsequent tasks. In Table 4.2, each row is derived from the corresponding row of the last table. The first column shows that the critical path if T1 is waited for before execution of T2 and the second column if the expected sequence is opposite. The last column shows an event-driven model.

Table 4.2 Effect of Various Scheduling

No.	*T3_T4*	*T4_T3*	*Event_driven*
1	13.8	14.4	13.8
2	15.5	15.6	15.5
3	18	18	18
4	15.4	17.7	15.4
5	18.3	16	16

The observations of interest from the experiments can be summarized as follows:

- When the difference in critical path compensates in an arm, the difference between the two scheduling schemes would be greater. Observations 4 and 5 present cases for the same.
- When the difference in critical path cumulates in an arm, there will not be any difference between the two scheduling schemes since the critical arm remains critical regardless of however it is scheduled.
- The event-driven scheduling always gives the same performance as the better of the two schemes. As the task graph becomes more complicated, there would be more possible schemes and only one of them would perform as well as the event-driven model.
- The scheduling schemes that are referred to above are mainly targeted at scheduling the various arms of the task graph in an integrated task-data parallel model. These strategies do not stem from a perspective of scheduling for load balancing.

The next experiment (see Table 4.3) was conducted to show the effect of exploiting data parallelism, task parallelism, and both task and data parallelism. For comparison, the sequential time of execution is also presented. The second column states the nature of the parallelism exploited: NOP, for No Parallelism exploited; DP for Data Parallelism exploited; TP for Task parallelism exploited; and TD for Task and Data Parallelism exploited. The other columns show the number of machines utilized for parallel computation and the time of completion of the problem. The split up of the time of completion is also shown.

The first row corresponds to the sequential execution of the problem. The split up of total time is the time taken for T1 to T4, respectively. The second and third rows present the results with data parallel execution on two and three nodes, respectively. The split up in this case is also the time taken for T1 to T4, respectively. The scaling down of the time of execution is accounted for by the data parallel execution of tasks. The fourth row shows the task parallel execution on two machines. The split up in this case is

Table 4.3 The Effect of Exploiting Task and Data Parallelism (Time in Minutes)

No.	*Parallelism*	*No. of machines*	*Time of completion*	*Split up*
1	NOP	1	74.2	23 + 24 + 12 + 15.2
2	DP	2	41.7	14 + 14.2 + 6.5 + 7
3	DP	3	26.3	8 + 9 + 4.5 + 4.7
4	TP	2	36	24 + 12
5	TD	4	19.8	(13.3, 12.7) + (6.5, 5.2)
6	TD	6	13.2	(8.4, 8.2, 8.0) + (4.3, 4.1, 4.8)

the time of tasks in the critical path. The last two runs employ both task and data parallelism with two nodes per task and three nodes per task, respectively. The split ups in these cases represent the time taken by data parallel sub-tasks of the tasks in the arm which proves to be the critical path.

The observations of interest from the experiments can be summarized as follows:

- The problem is a case where the task and data parallelism are complementary.
- The control parallelism in the problem saturates with the utilization of two nodes. This is because there are only two control parallel arms in the application.
- The data parallelism in the problem starts saturating with the utilization of three nodes. It could be seen from the third row that the granule size has reached around four minutes of execution time. Further sub-divisions for parallelism do not yield results because of the fixed time overheads of splitting the problem, migrating the code and arguments, compiling the code, and collecting the results.
- It can be seen that with the exploitation of both task and data parallelism, six nodes are utilized for parallel execution before the same granule size of four minutes is reached.

4.7 Guidelines for Composing User Programs

Composing a program over the platform involves translation of the task graph of the application into the final code. A task graph can be expressed as a directed graph with nodes representing tasks and links precedence relationships. A task could be a starting task, an intermediate task or the final task. Starting tasks are those which do not have to meet any pre-conditions for their execution. Intermediate tasks have precedence constraints to honour. The final task is one which is responsible for termination of the program.

The *Task begin* of starting tasks needs no *OnFinish* directives whereas it is required for the intermediate tasks and the *final task*. The final task should take care of the termination of the program, otherwise the program will wait in an infinite loop for further events to come. In cases where there are more than one final task, the methodology insists on keeping a single pseudo-final task which takes care of termination.

It was mentioned earlier that the expressibility provided employs block structuring of the code. A block designates a task and could contain other blocks in it. Blocks written inside a block are those which can execute only after the completion of the outer block. The complete precedence requirements, including the implications from the structure of the code, have to be specified.

Tasks which have more than one precedence relationship to be met could be written as a block inside any one of their parent blocks. In such cases, the choice of the parent block is left to the programmer's discretion. However, the placing of such blocks would not have any effect on the translated code. It should be noted that such tasks should not be placed in more than one parent block. Also, readability of the code can be enhanced by placing appropriate comments wherever such discretions are made.

While composing programs with existing modules, programs for each task could be available as individual files. In such cases, the structuring policy need not be strictly followed. The only modification that is to be done in such cases is to wrap the code for each task with task demarcating constructs, *Task begin* and *Task end*, along with their *OnFinish* directives.

The passing of arguments to tasks should be carried out keeping in mind the syntax, semantics and limitations of the call which supports it. Some systems are not designed for the tasks to take more than one stream as argument. In such cases, the programmer has to explicitly pack the argument streams before passing the argument, and then unpack it in the target task.

The *Register event* calls are inserted by the parser itself. The call has at least once semantics. A redundant insertion of the call by the programmer would be ignored.

While programming for anonymous execution, no assumption should be made about the underlying system. Although the portability of the system is provided, the portability of the migratable user program has to be ensured by the programmer himself.

4.8 Related Work

There have been a few attempts at integrating task and data parallelism in the literature. The notable ones include Opus [14], Fx [15], Data Parallel Orca [16] and Braid [17]. Opus integrates task parallel constructs into data parallel High Performance Fortran (HiPF). A task in Opus is defined as a data parallel program in execution. Due to this heavyweight notion of a task, inter-task communication is costly and hence, Opus is suited only to coarse-grained parallelism. Fx also adds task parallel constructs into HiPF. However, it uses directives to support task parallelism. It does not allow the arbitrary nesting of task and data parallelism. Data parallel Orca uses language constructs to integrate data parallelism into task parallel Orca. It has a limited notion of data parallelism as it does not support operations that use multiple arrays. This is because Orca applies operations only to single objects. Braid is a data parallel extension to task parallel Mentat. It uses annotations to determine which data an operation needs. The four models are cases of integrating task and data parallelism in specific languages. The proposed model, however, is a generalized methodology to integrate task parallelism into any data parallel language. The model targets coarse grain real task-data parallel computing on loaded NOWs. Furthermore, it supports the arbitrary nesting of task and data parallelism, unlike the existing models. Table 4.4 summarizes the comparison of our model with the existing works.

4.9 Future Work

A further challenge in transparent platforms for NOWs is to support communicating parallel tasks. The key issue in such an area is to provide message-passing abstractions in the premises of distribution transparency. Such an attempt needs to address problems with patterns in their process interaction. Optimizations possible with different interaction schemes could be explored.

A generic specification scheme for programs to specify the constraints on task/sub-task mapping could be explored. This, along with anonymous execution, will make it possible for the programmer to exploit the best of task and data parallelism. The constraints could be inflexible when a piece of code makes assumptions concerning the underlying system. The set of constraints could also include directives by the programmer of some hidden possibilities of optimizations that could be exploited.

An approach to characterize nodes in a pool of machines keeping in view the spectrum of distributed computations, could be attempted.

4.10 Appendix

This section describes the messages that are exchanged between the daemons and the user processes. Message exchanges occur between a user process and the *lc* on its machine, between *lc*s and the *sc*, and between an *lc* and the processes which have migrated to its machine for execution.

Table 4.4 Comparison of Various Task-data Parallel Integration Models

System	*Fx*	*Opus*	*Data parallel orea*	*Braid*	*Our model*
Aim	Integrate task parallelism into data parallel HPF	Integrate task parallelism into HPF	Integrate data parallelism into task parallel Orca	Integrate data parallelism into task parallel Mentat	Integrate task parallelism into *any* data parallel language
Basis of implementation	Compiler	Runtime System (RTS)	RTS	Annotation	RTS
Expressibility (exp)	Restricted form of task parallelism	Full exp	Restricted form of data parallelism	Full exp	Full exp
Communication between tasks	Shared address space	Shared object	Shared object	Shared object	Dependent on native data parallel language
Grain size	Fine	Coarse	Fine	Fine	Coarse

The messages received by the *lc* are as follows:

From the user program:

- Get load factor: Prior to the submission of a set of tasks, a user program requests the load.
- Information of a required number of nodes: In response to this, the *lc* forwards this message to the *sc*.
- Take work: This is a task submission message. The destination is indicated in the message itself, and this is followed by the string representing the RIBs source file name and the arguments to the task. The *lc* forwards this set of messages to the *sc* after recording the information required for recovery.
- Check results: This message is sent when the user program calls the *peek results()* function. The *lc* checks in its structures for the presence of the required results and sends a corresponding reply to the user program. The content of this reply is the return value of the function.
- Want results: This message is sent when the user program calls the *obtain results()* function call. In response to this, the *lc* checks in its structures for the presence of the required results and sends a corresponding reply to the user program. The function call is blocking and sends this message repeatedly after a waiting period, incorporating task re-submissions if necessary.
- User is terminating: This message says that the user program is terminating. In response to this, the *lc* invalidates the user-program-related information maintained in its structures.
- Register event: This message is used by the user program to inform the *lc* of the events of its interest. In response to this, the *lc* updates the event table corresponding to the user process.
- Register tasks: This message is used by the user program to declare a task as a collection of sub-tasks. In response to this, the *lc* updates the task table corresponding to the user process.

From the sc:

- Take load factor: This message is in response to an earlier *get load factor* message sent on behalf of a user program. It is followed by a message which represents the actual load factors. This set of messages is forwarded to the user program.
- Take work: This message represents a task submission. It is followed by the source file name for the task and the arguments to the task. The *lc* checks whether the said task is already running, i.e. whether the current message represents a re-submission. If not, or if the task failed, the *lc* compiles the source file and spawns a task process to execute the task.
- Take results: This message represents the results of an earlier submitted task. It is followed by the actual result. The *lc* stores the result in its structures, so that it can immediately respond to a *want results* or *check results* from the user program. Also, with each of the take results, the *lc* checks if some event of interest registered in the event table has occurred. If so, it informs the user process by generating an *event occurred* message.
- Generator failed: This message indicates that a particular user program has failed. The *lc* uses its recovery information to track all the tasks created by this particular user program and kills them, since they are no longer useful.
- LC is terminating: This message indicates that a particular *lc* has failed. The *lc* uses its recovery information to track all the tasks created by the user programs on the machine and kills them.
- Terminate: This message is sent by the *sc* asking *lc* to quit. In response to this, the *lc* quits.

From the task process:

- Take results: This is a message from the task process saying that it has completed the task assigned to it. This message is followed by the actual results of task execution, after which the task process exists. The *lc* forwards the result to the *sc*, after which it invalidates the recovery information maintained for the task.

- Give arguments: This message is sent by the task process as soon as it comes up. It is a request for the arguments of the task. The *lc* responds with a *take arguments* message followed by the actual arguments for the task.

The messages received by the *sc* are as follows:

- Get load factor: This messages is sent by an *lc* to the *sc* prior to the submission of tasks to the system. The purpose of this message is to ask for a certain number of machines in order to perform tasks. In response to this, the *sc* finds the requisite number of least loaded machines from the *sc* structures, orders their load factors in ascending order and sends it to the requesting *lc*.
- Take load factor: This message is sent by an *lc* to the *sc* to inform it of the current load factor on its machine. In response to this, the *sc* makes a record of this information in its structures.
- Take work: This message, initiated by a user program, is forwarded by an *lc* to the *sc*. It represents a task submission. It is followed by the string representing the RIB source file name and the arguments to the task. In response to this, the *sc* sends this set of messages to the appropriate *lc* as indicated in the first message.
- Get code: This message is sent from an *lc* to the *sc* and is meant for another *lc*. It represents a request for the source code for a migratable module, and will be needed if the two machines in question are not on the network file system. In response to this, the *sc* forwards this to the corresponding *lc*.
- Take code: This message is the counterpart of the previous message. The *sc* handles this message in the same way as the previous one.
- Take results: This message is sent by an *lc* to the *sc* and represents the results of a task submission. This is followed by a message containing raw data, which represents the actual results. In response to this, the *sc* forwards this set of messages to the appropriate *lc*.

- Generator failed: This message is sent by an *lc* to the *sc* informing it that a particular use program on its machine has failed. This is helpful because the tasks submitted by the user program no longer need to be executed, and hence can be killed, removing a possibly substantial workload. In response to this, the message is forwarded to the corresponding *lc*s.
- *lc* is terminating: This fact is recorded by the *sc* either when an *lc* terminates of its own accord or fails suddenly, or the network connection to it fails. In response to this, the *sc* broadcasts this message to all the other *lc*s.

All the messages that a user program receives are from *lc*. The messages received by the user program are as follows:

- Take load factor: This is in response to one of the earlier *get load factor* message generated by the user program. It gives the required machine indices and their load factors to the user program.
- Result availability: This is in response to a *check results* message generated by the user program. This message lets the user program know if the results are available or not.
- Take results: This is in response to a *want results* message generated by the user program. This message passes the results to the user program.
- Event occurred: This message is used to let the user program know the occurrence of an event of its interest. It specifies the event or events that have occurred. In response to this, the user process can determine its further course of computation.

References

1. Hatcher, P.J. and M.J. Quinn, *Data-parallel Programming on MIMD Computers*, The MIT Press, Cambridge, MA, 1991.
2. Kumar, V., A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing*, The Benjamin/Cummings Publishing Company Inc., 1994.

3. Seevers, B.K., M.J. Quinn and P.J. Hatcher, "A Parallel Programming Environment Supporting Multiple Data Parallel Modules", *SIGPLAN*, pp. 44–47, January 1993.
4. Gross, T., D.R. O'Hallaron and J. Subhlok, "Task Parallelism in a High-performance Fortran Framework", *IEEE Parallel and Distributed Technology*, Vol. 2, No. 3, pp. 16–26, 1994.
5. McRae, G., A. Russell and R. Harley, *CIT Photochemical Airshed Model: Systems Manual,* 1992.
6. Chapman, B., H. Zima and P. Mehnotna, "Extending HPF for Advanced Data Parallel Applications", *IEEE Parallel and Distributed Technology,* Vol. 2, No. 3, pp. 59–70, 1994.
7. Foster, I., "Task Parallelism and High-performance Languages", *IEEE Parallel and Distributed Technology*, Vol. 2, No. 3, pp. 27–36, 1994.
8. Bal, H.E. and M. Haines, "Approaches for Integrating Task and Data Parallelism", *IEEE Concurrency,* pp. 74–84, 1998.
9. Joshi, R.K. and D.J. Ram, "Anonymous Remote Computing: A Paradigm for Parallel Programming on Interconnected Workstations", *IEEE Transactions on Software Engineering,* Vol. 25, No. 1, pp. 75–90, 1999.
10. Parthasarathy, R., "Designing a Robust Runtime System for ARC", *Project Report, Acc. No. 97-BT-04*, Department of Computer Science and Engineering, IIT, Chennai, 1997.
11. Ramachandran, R.P., M.S. Zilovic and R.J. Mammone, "A Comparative Study of Robust LP Analysis Methods with Applications to Speaker Identification", *IEEE Transactions on Speech, Audio Processing*, Vol. 3, pp. 117–125, 1995.
12. Reynolds, D.A. and R.C. Rose, "Robust Text-independent Speaker Identification Using Gaussian Mixture Speaker Models", *IEEE Transactions on Speech, Audio Processing,* Vol. 3, pp. 72–83, 1995.
13. Chandrasekhar, C., B. Yegnanarayana and R. Sundar, "A Constraint Satisfaction Model for Recognition of Stop Consonant–Vowel (SCV) Utterances in Indian Languages",

Proceedings of the International Conference on Communication Technologies (CT-96), Indian Institute of Science, Bangalore, pp. 134–139, December 1996.

14. Chapman, B., *et al.*, "Opus: A Coordination Language for Multi-disciplinary Applications", *Scientific Programming*, Vol. 6, No. 2, 1997.
15. Sublhok, J. and B. Yang, "A New Model for Integrated Nested Task and Data Parallel Programming", *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, ACM Press, pp. 1–12, 1996.
16. Hassen, S.B. and H.E. Bal, "Integrating Task and Data Parallelism Using Shared Objects", *Proceedings of the 10th ACM International Conference on Supercomputing*, ACM Press, pp. 317–324, 1996.
17. West, E.A. and A.S. Grimshaw, "Braid: Integrating Task and Data Parallelism", *Proceedings of the Frontiers 1995: Fifth Symp. Frontiers of Massively Parallel Computation*, IEEE CS Press, pp. 211–219, 1995.

Chapter 5

Anonymous Remote Computing and Communication Model*

5.1 Introduction

Harnessing the potential parallel computing power of a non-dedicated cluster of workstations is an arduous task. The set of nodes which comprise the cluster of workstations changes over time. The nodes, as well as the communication links, are liable to failure. The nodes could exhibit heterogeneity in architecture, processing power and operating systems. Parallel programs on such non-dedicated nodes of the network would need to co-exist with the regular load on these nodes. This results in an uneven load across the nodes and changes in load characteristics with time.

Prior research has reported an effective parallel solution to problems without inter-task communication. Condor [1], Piranha [2], ADM [3], ARC [4], Sprite [5], V system [6], and EMPS [7] are some of the notable efforts. However, most of them do not address inter-task communication.

The effective parallel solution of problems on NOWs require the runtime selection of nodes. The granularity of individual sub-tasks may have to be deferred until runtime for load balancing. Dynamic schemes may be needed to make the programs resilient to changing conditions. Support for inter-task communication in

*Binu K. Johnson, R. Karthikeyan, D. Janakiram high-level parallel programming platforms which support runtime and dynamic policies, leads to several issues. Some of these issues are as follows:

- In ARC, the node to which a sub-task is migrated is decided at runtime. The nodes remain anonymous to the user program which initiates the migration. Thus, a sub-task would be unaware of the location of other sub-tasks in order to communicate with them.
- Another issue pertains to the dynamic schemes employed by the platforms. The system support may migrate an already running task, reshuffle the load allotted to individual sub-tasks, etc. Such dynamic policies detach processes from specific nodes. Hence, communication primitives which assume the location of processes are unsuitable.

Transparent inter-task communication will facilitate a number of application domains to exploit the parallel computing power of clusters of workstations.

Some of these application domains are as follows:

- Iterative grid computations comprise a large class of engineering applications. When the domain of computation of an iterative grid computation problem is divided, the sub-domains will need to exchange their boundary values. Grid computations are used to solve problems, such as elliptical partial differential equations by finite differences [8].
- Parallel solutions of the class of sub-optimal algorithms like simulated annealing are discussed in [9]. The problem partitioning adopted requires the sub-tasks to exchange their intermediate results.
- Some problems can be partitioned for parallel solution as a network of filters. Networks of filters can be used to solve a variety of programming problems. Reference [10] describes a prime number sieve and a matrix multiplication network using this pattern. Such problems would also require the communication of intermediate results.

The current work explores the transparent programmability of communicating parallel tasks on loaded heterogeneous workstations.

5.2 Location-independent Inter-task Communication with DP

Distributed Pipes (DP) is a model for transparent programming of communicating parallel tasks. It addresses issues specific to parallel programming on NOWs. DP provides a set of high-level location-transparent communication primitives, which support data flow between processes that are independent of their location. This enables the model to accommodate anonymous migration of communicating parallel tasks.

In the model, the communication channels between the nodes of a network are considered as global entities. The information pertaining to them is maintained globally on a designated node. A communication channel is created or deleted at runtime. Communicating parallel tasks created at runtime can be connected by using DP. The high-level abstractions of DP provide an elegant set of programming interfaces that are free from low-level network details. This contributes to the readability and maintainability of the code. Programs in the model are not tied up to specific nodes. Hence, it accommodates a changing pool of workstations, and relieves the programmer from the task of programming for specific machines, thereby rendering the resultant code portable to a different network. The DP provides a uniform set of interfaces for communication across heterogeneous nodes. This addresses heterogeneity among the nodes in both architecture and the operating system. DP uses the external data representation to handle heterogeneity. The programming level abstractions of DP wrap TCP abstractions. Message sizes exceeding the size limit imposed by TCP are handled transparently by splitting and coalescing the message appropriately.

5.3 DP Model of Iterative Grid Computations

Figure 5.1 shows the nature of a typical iterative grid computation. The iterative marching in space and time dimensions are shown in the illustration on the left in Fig. 5.1. The expanded grid on the

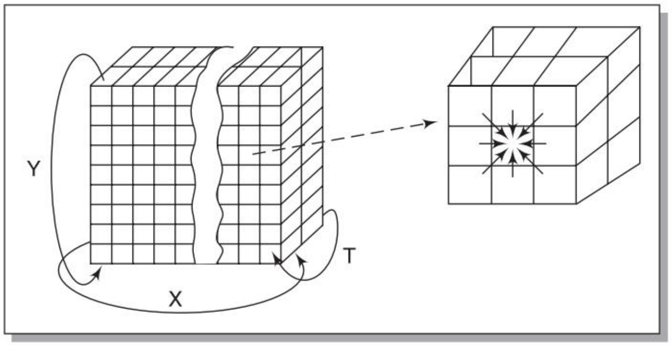


FIG. 5.1

Grid Computation Problem

right of Fig. 5.1 shows the boundary value exchanges. The typical program structure of the problem for sequential execution is as follows:

Pseudo Code 1: Program Structure of typical iterative grid computations

```
FOR Time = StartTime TO EndTime
  FOR XAxis = StartX TO EndX
    FOR YAxis = StartY TO EndY
      UserDefinedFunction()
    ENDFOR
  ENDFOR
ENDFOR
```

The outer loop of the pseudo code marches in time and the inner loops march in each dimension of space.

5.3.1 The Model

The model employs a master-worker model of computation. The program for the model consists of a master process and several worker processes. The master process is the process which initiates the computation. Worker processes are spawned on the nodes which participate in parallel computation. Worker processes are called Iterative Grid Modules (IGMs).

The model accomplishes parallel execution of the problem by domain decomposition. Each IGM is allotted a sub-domain of computation. The boundary value exchange of values between IGMs is effected through DPs. IGMs return the results of their computation to the master process. The model permits communication between anonymously migrated IGMs.

In the model, the system handles domain decomposition, selection of least loaded nodes, load balanced division of tasks, anonymous migration of IGMs, transparent laying of communication primitives between IGMs, result collection, and aspects related to fault tolerance.

The model offers various advantages. The programs in the model are not tied up to specific machines. Such programs can accommodate a changing pool of workstations. It also makes the programs portable to a different network. The number of IGMs is not decided *a priori*. Hence, the system can utilize the optimum number of nodes according to the runtime conditions. The programs written for the model can tolerate a heterogeneous collection of unevenly loaded workstations. The programs in the model are devoid of any underlying network code. This results in greater readability of the program and, hence, in greater maintainability.

Figure 5.2 illustrates the parallel solution of a grid computation problem using the model. In Fig. 5.2, thick circles represent IGMs, the thin circle represents the master process, ellipses represent runtime daemons, thick lines represent TCP connections, thin lines represent Unix Domain socket connections, and dashed lines depict DPs between IGMs. Grid Computation Tasks, (GCTs) represent IGMs and GCP (Grid Computation Problem) represents the Master Process.

5.3.1.1 Initialization

The master process and IGMs need to register with the system in order to avail of system services. The master process registers

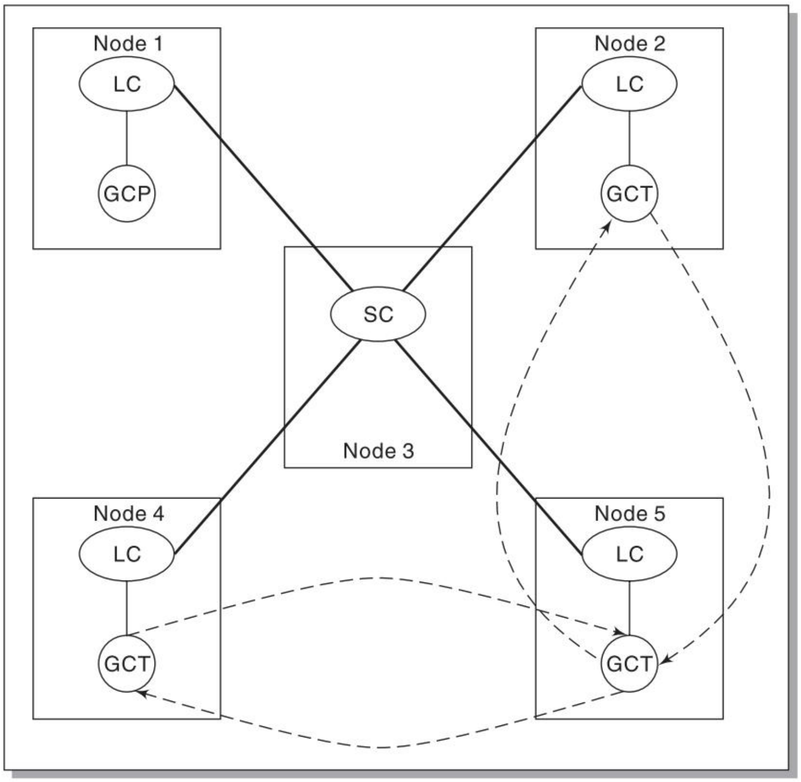


FIG. 5.2

Grid Computation Problem System Structure

with the system by using the call *InitializeWork()*. Upon completion, a complementary call *CloseWork()* is used.

IGMs register with the system by using the call *InitializeIGM()*. Upon completion, a complementary call *CloseIGM()* is used.

5.3.1.2 *Domain Decomposition*

In the model, the master process sends the grid information to the system. The system decides the optimum number of IGMs to be employed, the granularity of computation to be allotted to individual IGMs, and the nodes to be assigned for each IGM.

The master process gathers this information from the system and packs initial data for individual IGMs to migrate the IGMs and to create channels that collect results from each IGM.

The master process provides the grid information to the system by using the call *SendGridInfo()*. Similarly, it obtains the number of IGMs employed by using the call *ObtainNumberOfSplits()*. The granularity of each IGM is obtained by using the call *ObtainSplitInfo()*.

A brief description of the calls is given below.

- *int SendGridInfo (int WorkId, int SpaceInX, int SpaceInY, int SpaceInZ, int History, int SplitDirection).*

  SendGridInfo() sends the grid information of a grid computation work to the system.

  WorkId is the index by which the system identifies a grid computation work.

  SpaceInX, *SpaceInY*, and *SpaceInZ* are the number of grids in X, Y, and Z dimensions of space.

  History specifies the number of previous time slices to be stored.

  SplitDirection specifies the direction of the split.

- *int ObtainNumberOfSplits(int WorkId).*

  ObtainNumberOfSplits() collects the number of IGMs for the grid computation work denoted by the *WorkId.*

- *int ObtainSplitInfo(int WorkId, int SplitId, int* Start, int* Total).*

  ObtainSplitInfo() gathers information related to a split of the work. The starting grid of the sub-domain and the number of grids in the sub-domain are stored at the addresses pointed to by *Start* and *Total*, respectively.

5.3.1.3 *Load Balancing*

The system gathers availability and load information of the nodes in the network in order to decide the optimum number of IGMs to be employed and their individual granularities. Machines with load indices higher than a designated value are ignored. The domain of computation is sub-divided among the other machines. The granularity of individual sub-domains depends upon the load ratio of the machine. The load balancing scheme is discussed in Section 5.5.

Our approach to load balancing offers several advantages. The actual means of gathering and interpreting load information on the participating machines are hidden from the programmer. In a heterogeneous collection of workstations, the processing power of individual nodes is also used for load balancing. In our approach, the programmer is relieved of the task of specifying the ratio of the processing power in a collection of heterogeneous nodes. The load balancing scheme may have to be altered to accommodate different types of nodes or to prune the load interpretation mechanism. In such cases, the user programs need not be modified in order to change the load balancing scheme.

Figure 5.3 illustrates the load balanced division of the grid computation work into IGMs of different granularities on a heterogeneous collection of unevenly loaded workstations. The DPs connecting them are also shown.

5.3.1.4 *Anonymous Migration of Sub-tasks*

The anonymous migration of an IGM is initiated when the master process invokes the *Migrate()* call. However, the master process does not furnish any machine specific arguments to the *Migrate()* call. The realization of anonymous migration is discussed in Section 5.4. The information required for an IGM to initialize its data structures as well as the initial data for the IGM are the parameters to the call. These are retained with the local *lc* of an IGM until the IGM claims them. The syntax and semantics of the call are given below.

- *int Migrate(int WorkId, int SplitId, char* MigrateFile, int DataType, void* Data, int SpaceInX, int SpaceInY, int SpaceInZ, int History, char* ResultPipe).*

Migrate() migrates the code for an IGM and provides it with initial data. The data consists of the number of grid points in X,

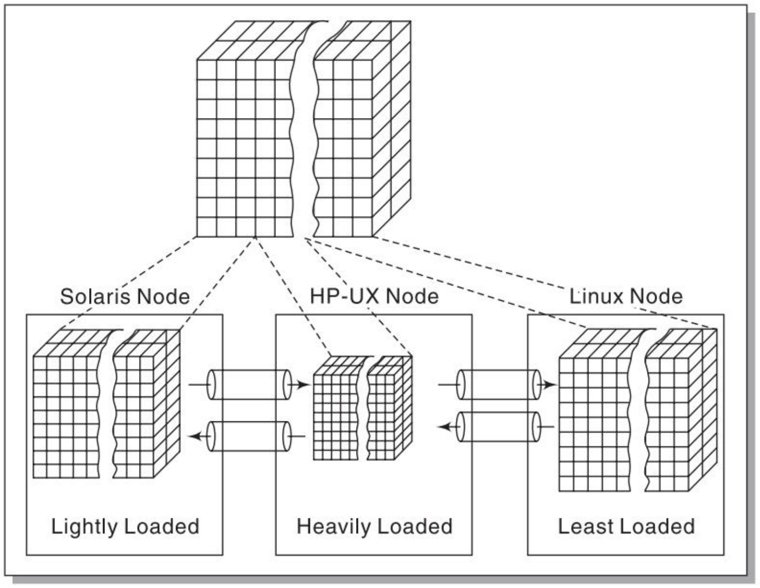


FIG. 5.3

Load Balancing Mechanism

Y, and Z dimensions of space. *ResultPipe* is the name of the DP to which the IGM writes its results.

5.3.1.5 *Information Gathering by IGMs*

The *lc* which collects anonymously migrated IGMs, compiles the IGM code and spawns the IGM process. An IGM process has to initialize its data structures to hold the initial data, and collect the initial data with which to begin computation, the position of the IGM, and the name of the Result Pipe to write its result. The size of the initial data is required to initialize the data structures. This is facilitated by the call *ObtainTask-GridInfo()*. After initializing the data structures, the IGM collects the initial data by invoking the call *ObtainTask-Data()*. The call *ObtainTaskMachineInfo()* provides the position of the IGM process and the name of the Result Pipe to be opened.

In our approach, the information pertaining to an IGM is not tied up with the code of the IGM. At runtime, the system provides information relevant to individual IGMs. The syntax and semantics of the calls are given below.

- *int ObtainTaskGridInfo(int* SpaceInX, int* SpaceInY, int* SpaceInZ, int* History).*

  ObtainTaskGridInfo() provides the number of grid points in X, Y, and Z dimensions of space.

- *int ObtainTaskData(void* Data, int SpaceInX, int SpaceInY, int SpaceInZ, int History).*

  ObtainTaskData() stores the initial data matrix at the address pointed to by Data.

- *int ObtainTaskMachineInfo(int* WhichMachine, char* ResultPipeName).*

ObtainTaskMachineInfo() stores the location of the IGM in the grid computation work at the address pointed to by *WhichMachine.*

5.3.1.6 *Transparent Communication*

Each IGM has to communicate with its neighbouring IGMs to exchange boundary values. Support for communication between IGMs brings with it two issues. Since the IGMs are migrated to anonymous nodes, an IGM will not know the location of its neighbouring IGMs. The second issue pertains to the position of an IGM. The number of neighbours of an IGM depends upon the position of the IGM. Hence, the number of DPs to be opened by the IGM cannot be known until runtime.

In the model, an IGM collects information about its neighbours and the number of DPs to be opened at runtime.

This is facilitated by the calls

ObtainTaskOpenPipeNames..

and

ObtainTaskNoOfPipesToBeOpened..:

Following is a description of the call:

- *int ObtainTaskOpenPipeNames(char** PipeNames, int* AccessMode, int NoOfOpenPipes).*
- ObtainTaskOpenPipeNames() provides the number, names and access modes of the Distributed Pipes to be opened.
- int ObtainTaskNoOfPipesToBeOpened(int* NoOfOpen-Pipes).

  ObtainTaskNoOfPipesToBeOpened() stores the number of Distributed Pipes to be opened at the address pointed to by NoOfOpenPipes.

5.4 Design and Implementation of Distributed Pipes

5.4.1 Runtime Support

The runtime support consists of an *lc* daemon running on each node participating in parallel computation and a *sc* daemon on a designated node.

Figure 5.4 illustrates the overall structure of the system. Circles represent the user processes, ellipses represent runtime daemons, thick lines represent TCP sockets, thin lines represent Unix Domain sockets, and dashed lines represent Distributed Pipes.

5.4.1.1 Local Coordinator (lc)

The *lc* runs on each node that participates in parallel computation. The *lc* services requests generated by user processes on its node. Also, it maintains information required to coordinate the user processes.

The *lc* maintains two tables to support bare DP services, namely, the User Process Information Table (UPT) and the User Processes Blocked for Write Table (UPBWT). The UPT maintains information pertaining to user processes which have registered with the *lc* on its node. UPBWT keeps track of processes which

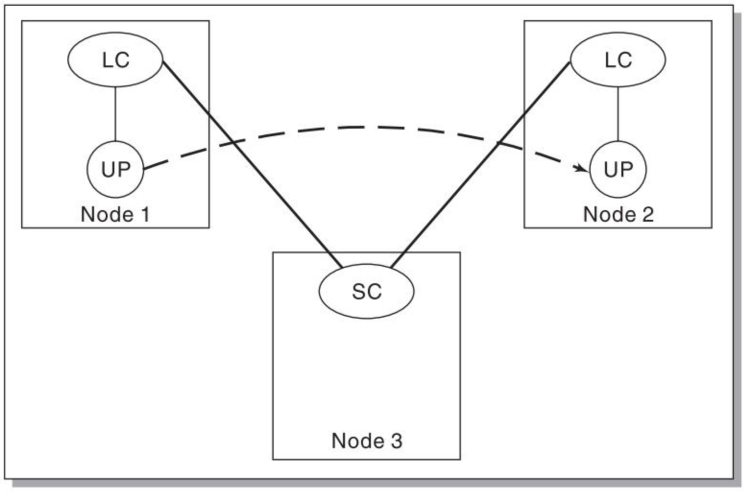


FIG. 5.4

System Structure

have opened a DP in write mode and have not been opened by any other process for reading. The writing process is blocked until the DP is opened by some other process in read mode. The *lc* maintains the table in order to inform the blocked processes when another process opens the DP in read mode.

In order to support grid computations, the *lc* maintains a Grid Computation Task Submitted Table (GCTST). The *lc* uses the GCTST to service the requests of a task. The table is indexed by the process id of the task. The GCTST is updated either when a new task is submitted or when an already submitted task terminates. If the service needs additional parameters, the *lc* forwards the information to the *sc*.

The FSM of *lc* is given in Fig. 5.5. In the INIT state, the *lc* initializes its data structures and cleans the auxiliary system files. The *lc* establishes a TCP connection with the *sc* and registers with the *sc*. In the LISTEN state, the *lc* waits for messages from the *sc* or any user process. When it receives a message from the *sc*, it changes its state to SC Msg RECVD and services the message.

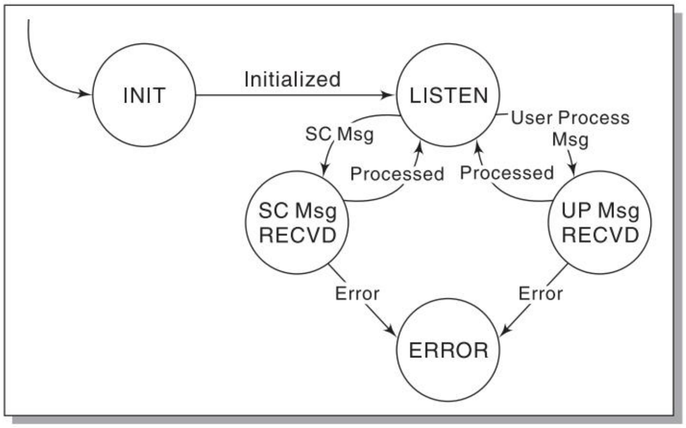


Fig. 5.5

FSM of Local Coordinate (*lc*)

When it receives a message from a user process, it changes its state to UP Msg RECVD and services the request.

The initial communication between a process and the *lc* is through a known common channel. This is required for a user process to register with the *lc*. User processes which register with the *lc* are given exclusive communication channels for subsequent communication.

5.4.1.2 *System Coordinator (sc)*

The *sc* coordinates the *lcs* in the pool. Also, the *sc* keeps track of individual *lcs* and facilitates communication between them. The *sc* is connected to *lcs* through TCP sockets. The *sc* maintains TCP socket descriptors which connect it to individual *lcs*.

The *sc* maintains two tables, namely, the Distributed Pipes Table (DPT) and the Local Coordinators Table (LCT). The DPT keeps track of the DP channels. The table is updated when a DP is created, opened, closed, or deleted. When a process opens a DP to write to, before the pipe is opened for reading, the corresponding *lc* information is also stored in the DPT. Thus, the process can be intimated when some other process opens the DP in read mode.

The LCT keeps track of the *lcs* in the system. The table is updated when a new *lc* joins the pool or when an existing *lc* leaves the pool.

The *sc* maintains two additional tables in order to support grid computations, namely, the Grid Computation Work Table (GCWT) and the Grid Computation Task Table (GCTT). The GCWT maintains information pertaining to grid computation work that is submitted to the *sc*. It is updated either when a work is submitted to the *sc* or when a work is completed. The GCTT maintains information pertaining to individual tasks that constitute the grid computation work. The table is updated when the work is sub-divided into tasks, when a task begins execution, or when a task terminates. The GCTT is a part of the GCWT.

The FSM of *sc* is given in Fig. 5.6. In the INIT state, the *sc* initializes its data structures and cleans the auxiliary system files. In the LISTEN state, the *sc* polls for connection requests from the *lcs*. When a connection request from an *lc* is received, it registers the *lc* with the system and establishes a TCP socket connection between them. It then listens for messages from the registered *lcs* on exclusive channels, and continues to listen for new connection requests. When a message from an *lc* is received, it changes its state to LC Msg RECVD and processes the message. Once the message is processed, it returns to the LISTEN state.

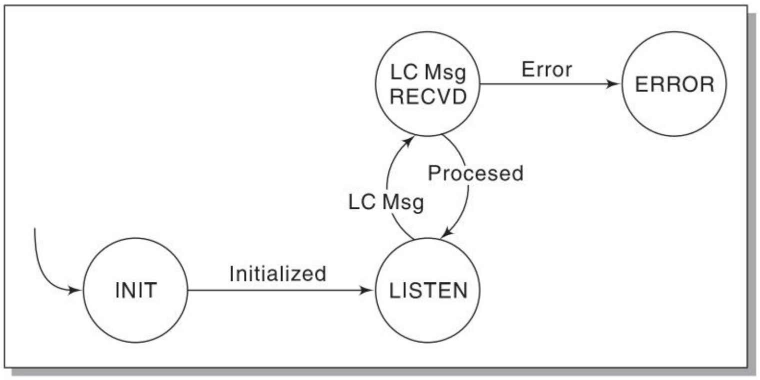


FIG. 5.6

FSM of System Coordinator (*sc*)

5.4.2 Functional Library Support

The functional library support consists of services to support location-transparent communication with DPs and services to support the DP model of iterative grid computations. Variants of the calls are provided to support communication across heterogeneous architectures, by utilizing the external data representation. The library is built over TCP and Unix domain stream protocol.

5.4.2.1 Basic Distributed Pipe (DP) Services

The following are the basic DP services:

- *int CreateDistPipe(char* PipeName).*

  CreateDistPipe() initiates a message to the *sc* through the *lc* running on its machine. The *sc* creates the DP if another channel with the same name does not exist and makes an entry in the DPT.

- *int OpenDistPipe(char* PipeName, int AccessMode).*

  OpenDistPipe() initiates a message to the *sc* through the *lc* running on its machine with the name of a DP as a parameter. The *sc* completes the message sequence by informing the user process if the DP is created or not.

  Corresponding to an open request in Write Mode, a TCP socket is created and another message sequence is initiated by sending a message to *sc* through *lc*. The message contains the TCP socket descriptor, access mode, and process id of the requesting process. The *sc* updates DPT with this information. If the DP is already opened by another process in Read Mode, the information of the read process is returned to the caller. The open call uses this information to connect to the read process. If the DP is not opened for reading, it causes the update of DPT at *sc* and UPBWT at *lc*. Subsequently, the call blocks until it receives a message from the *lc* intimating the information of read process.

  Corresponding to an open request in Read Mode, a TCP socket is created and bound to a local port. A message is generated to the *sc* to update the DPT with the TCP socket descriptor, port number, access mode, and the process id. Further, the *sc* intimates the user processes which have requested to open the channel in Write Mode with the details of the read process. The call listens on the TCP socket for connection requests.

- *int ReadDistPipe(int PipeDescriptor, char* Buffer, int BufferSize).*

  A *ReadDistPipe()* call translates to the read system call. It reads from the socket descriptor for the DP descriptor. The PipeDescriptor returned is the actual socket descriptor in order to make it a direct translation. Hence, the read call does not cause any overheads. The call handles message sizes exceeding the limits of TCP messages.

- *int WriteDistPipe(int PipeDescriptor, char* Buffer, int BufferSize).*

  A *WriteDistPipe()* call translates to the write system call. It writes to the socket descriptor for the DP descriptor. The PipeDescriptor returned is the actual socket descriptor in order to make it a direct translation. Hence, the write call does not cause any overheads. The call also handles message sizes exceeding the limits of TCP messages.

- *int CloseDistPipe(int PipeDescriptor)*

  CloseDistPipe() call translates to the close system call. It closes the socket descriptor for the DP descriptor. Further, the call initiates a message to the *sc* through the *lc* with the name of the DP and process id as parameters. This causes the DPT table at the *sc* to be updated.

- *int DeleteDistPipe(char* PipeName).*

  DeleteDistPipe() initiates a message to the *sc* through *lc* with the name of the DP as an argument. In response to the message, the *sc* deletes the corresponding entry in DPT and returns the deletion status to the call.

5.4.2.2 *Overhead of Interfaces*

The overhead of each call is caused by the message sequences initiated by the call. The calls *CreateDistPipe*, *CloseDistPipe*, and

DeleteDistPipe result in a message to the *lc* on the node, a message from the *lc* to the *sc* over the network, a reply from the *sc* to the *lc* over the network, and a message from the *lc* back to the user process. The call *OpenDistPipe* constitutes two such message sequences and, hence, twice the overhead. The typical size of data-packets exchanged is around 100 bytes. The round trip time of communication over network could range from 0.4 milliseconds (ms) to a few milliseconds. Typically, the average round trip time is less than two milliseconds. However, these calls are used only once during the lifetime of a DP. Hence, these overheads become insignificant. The calls *ReadDistPipe* and *WriteDistPipe* are directly translated to the underlying system call. Hence, they do not incur any overheads. These calls are used many times during the lifetime of a DP.

5.4.2.3 *Extended Services for IGC*

The extended sevices for IGC are as follows:

- *int InitializeGridComputationWork().*

  InitializeGridComputationWork initiates a message to the *sc* through *lc*. The *sc* creates an entry for the work in GCWT and returns the *WorkId*.

- *int SendGridInfo(int WorkId, int SpaceInX, int SpaceInY, int History, int SplitDirection).*

  SendGridInfo initiates a message to the *sc* through *lc*.

  The message carries the arguments to the call. The *sc* updates the information in GCWT and returns the updated status.

- *int ObtainNumberOfSplits(int WorkId).*

  ObtainNumberOfSplits initiates a message to the *sc* through *lc*. The *sc*, in response to the message, collects the load information of all machines from the corresponding *lcs*. This information is used by the *sc* to split the work. Further, the *sc* creates a new entry in GCTT to store the information about the split and returns the number of splits.

- *int ObtainSplitInfo(int WorkId, int SplitId, int *Start, int *Total).*

  ObtainSplitInfo initiates a message to the *sc* through the *lc*. The *sc* gathers the information from GCTT and returns the starting and total number of grids for the split.

- *int Migrate(int WorkId, int SplitId, char *MigrateFile, int DataType, void *Data, int SpaceInX, int SpaceInY, int History, char *ResultPipe).*

  Migrate initiates a message to the *sc* through *lc* with its WorkId and SplitId. The *sc* gathers the information of *lcs* from GCWT and GCTT and sends messages to them. In response to the message, each *lc* creates a TCP socket, binds the TCP socket to a local port, and listens on it. Also, the port numbers are returned to the *sc*. The *sc* passes the collected information to the *lc* which initiated the migration. The *lc* which initiated the migration makes a TCP connection and transfers the code, data, and result DP name to the other *lc*. The GCTST of the migrated *lc* is updated by using this information. Once the migration is over, the TCP connection is closed and the *sc* is informed.

- *int CloseGridComputationWork(int WorkId).*

  CloseGridComputationWork initiates a message to the *sc* through *lc* with *WorkId*. In response to the message, the *sc* purges the corresponding entry from GCWT.

- *int InitializeGridComputationTask.*

  InitializeGridComputationTask initiates a message to the *sc* through the *lc*. In response to the message, the *sc* updates GCTT with the new task entry and returns *TaskId*.

- *int ObtainTaskGridInfo(int *SpaceInX, int *SpaceInY, int *History).*

  ObtainTaskGridInfo initiates a message to the *sc* through the *lc*. The message contains the process id of the task. The *lc* collects the relevant information from the GCTST.

- *int ObtainTaskMachineInfo(int *WhichMachine, char *ResultPipe Name).*

ObtainTaskMachineInfo initiates a message to the *sc* through the *lc*. The *sc* returns the position of the sub-domain for which the task is responsible to the *lc*. The *lc* returns this information and the *ResultPipeName* to the task.

- *int ObtainTaskData(void *Data, int SpaceInX, int SpaceInY, int History).*

  ObtainTaskData initiates a message to the *lc*. The *lc* gathers the information from GCTST and returns it to the task.

- *int ObtainTaskNoOfPipesToBeOpened(int *NoOf OpenPipes).*

  ObtainTaskNoOfPipesToBeOpened initiates a message to the *sc* through the *lc*. The process id of the task is passed along the message. The *sc* gathers information from the GCWT and returns the number of DPs to be opened by the task.

- *int ObtainTaskOpenPipeNames(char **PipeNames, int *AccessMode, int NumberOfOpenPipes).*

  ObtainTaskOpenPipeNames initiates a message to the *sc* through the *lc*. The process id is passed along with the message. In response to the message, the *sc* returns the names of DPs to be opened and their Access Modes.

- *int CloseGridComputationTask(int TaskId).*

  CloseGridComputationTask initiates a message to the *lc*. The *lc* deletes the corresponding entry from the GCTST and forwards the message to the *sc*. In response to the message, the *sc* updates GCWT.

5.5 Case Study

Iterative grid computations comprise a large class of engineering applications. This class of applications exhibit a pattern in their process interaction [11]. Steady State Equilibrium Problem is considered for our case study. The problem computes intermediate temperature flux distribution of a rod whose both ends are kept in constant temperature baths. Similar computational problems exist in many engineering disciplines to compute pressure distribution, composition distribution, etc.

5.5.1 Problem Description

The problem iteratively computes the temperature values at each grid point. Each iteration computes the function for a time slice. The temperature of a grid at a time slice is influenced by the temperature of the grid in the previous time slice along with the temperature of adjacent grids in the previous time slice. This accounts for the temperature flux by conduction. The problem considers the temperature flux in only one dimension. The points at which temperature is to be computed are evenly spaced. Also, the temperature of a grid point is evaluated at regular intervals in time.

The data dependency among adjacent grids is shown in equation (1).

$$T_{g,t} = f(T_{g-1,t-1},\ T_{g,t-1},\ T_{g+1,t-1}) \qquad \text{(equation 1)}$$

where $T_{i,j}$ is the temperature of grid i during the time slice j.

Temperature values of the fixed temperature baths, length and distance between adjacent grid points, and time interval between two successive computations are the data furnished at the beginning of the run. The equations that characterize the flow of temperature are space-time domain equations.

5.5.2 DP Model of Computation

In the model, the problem is expressed as a master process and several worker processes. A worker process is responsible for computation over a sub-domain. The grain size of the sub-domain allotted to a worker process depends upon the load ratio of the node on which it executes. The master process initiates the computation. The master process also communicates with the system to coordinate the parallel computation.

5.5.2.1 The Master Process

The master process initiates the parallel computation by initializing the grid computation work with the system. The grid information of the problem is sent to the system. The system decides the number of worker processes, their individual granularities, and the machines on which they execute. It has been mentioned earlier that availability and load information of the machines in the pool influences this decision. The master process collects the number of worker processes that constitute the computation. Further, the master process collects the information of each split from the system. This is used by the master process to construct data-packets for each worker process. The Result Pipe for each worker process is created and migrated. Further, the master process opens the Result Pipes and waits for the results. The Result Pipes are closed and deleted after the results are collected. A sample code of the master process is given below (Pseudo Code 2).

Pseudo Code 2: Sample Program of the Master Process

```
int main ()
{
  ...
  WorkID = InitializeGridComputationWork();
  ...
  SendGridInfo(WorkID, NoOfRows, NoOfCols,
  1, SPLIT_COLUMNS);
  . . .
  Number of Splits = ObtainNumberOfSplits
  (WorkID);
  ...
  for(SplitId = 0; SplitId < NumberofSplits;
  SplitId++)
    {
      ObtainSplitInfo(WorkID, SplitId,
      andStart, andTotal);
      ...
      MakeDataPacket(Matrix, NoOfRows, Start,
      Total, 1, SPLIT_COLUMN, DataPacket);
      ...
      ResultPipe = HostName.“Result”.WorkID.
      SplitIndex;
      CreateDistPipe(ResultPipe);
      Migrate (WorkID, SplitIndex,
      MigrateFile, FLOAT_TYPE, DataPacket,
      ResultPipe);
    }
  ...
  for(SplitId = 0; SplitId < NumberofSplits;
  SplitId++)
    {
      ResultPipe = HostName.“Result”.WorkID.
      SplitId;
      PipeFd[SplitId] = OpenDistPipe(Result
      Pipe, READ_MODE);
    }
  for(SplitId = 0; SplitId < NumberofSplits
  SplitId++)
    {
  ReadSignedCharacter(PipeFd[SplitId],
  andMoreResult, sizeof(MoreResult),
  CONVERT_XDR);
     while(MoreResult)
      {
        ReadFloat(PipeFd[SplitId], andResult
        sizeof(Result), CONVERT_XDR);
        ReadSignedCharacter(PipeFd[SplitId],
        andMoreResult,sizeof(MoreResult),
        CONVERT_XDR);
      }
    }
  for(SplitId = 0; SplitId < NumberofSplits;
  SplitId++)
```

```
        {
          CloseDistPipe(PipeFd[SplitId]);
          ResultPipe = HostName.“Result”.WorkID.
          SplitId;
          DeleteDistPipe(ResultPipe);
        }
      CloseGridComputationWork(WorkID);
      return 1;
    }
```

5.5.2.2 *Iterative Grid Module (IGM)*

IGMs are migrated to anonymous remote nodes for execution. An IGM starts its execution by initializing itself with its local *lc*. IGMs collect information regarding the sub-domain allotted to them from the system. Similarly, IGMs collect the relevant information from the system by using the services provided. Pseudo Code 3 shows the system services availed by an IGM and the corresponding course of action taken.

Pseudo Code 3: Sample Program of an IGM

```
int main()
{
  ...
  TaskId = InitializeGridComputationTask();
  ...
  ObtainTaskGridInfo(andNoOfRows, andNoOfCols,
  andDepth, andSplitDirection);
  ObtainTaskMachineInfo(andWhichMachine,
  ResultPipeName);
  ...
  OpenDistPipe(ResultPipeName, PIPE_READ);
  ...
  ObtainTaskData(Data);
  ...
```

```
   ObtainTaskNumberofPipesToBeCreatedAnd
   Opened(andNoOfCreatePipe, &NoOfOpenPipes);
   ...
   ObtainTaskCreatePipeNames(CreatePipes,
   NoOfCreatePipe);
   for(PipeIndex = 0;
   PipeIndex < NoOfCreatePipes; PipeIndex++)
     {
       CreateDistPipe(CreatePipes [PipeIndex]);
     }
   ...
   ObtainTaskOpenPipeNames(OpenPipes,
   AccessModes, NoOfOpenPipes);
   for(PipeIndex = 0; PipeIndex < NoOfOpen
   Pipes; PipeIndex++)
     {
       PipeFd[PipeIndex] = OpenDistPipe(Open
         Pipes[PipeIndex], AccessModes
   [PipeIndex]);
     }
   ...
   if (WhichMachine ! = LAST_GCTM)
     {
   WriteFloat(NextMachineWd, andPrevTimePrev
   Grid, sizeof(float), 1, CONVERT_XDR);
   }
 if (WhichMachine ! = FIRST_GCTM)
   {
     WriteFloat(PrevMachineWd, andPrevTimeNext
     Grid, sizeof(float), 1, CONVER_XDR);
   }
 for(Time = 0; Time < MAX_TIME; Time++)
   {
     if (WhichMachine ! = FIRST_GCTM)
```

```
        {
          ReadFloat(PrevMachineRd, andPrevTime
          PrevGrid, sizeof(float), 1, CONVERT_
          XDR);
          WriteFloat(PrevMachineWd, andPrevTime
          NextGrid, sizeof(float), 1, CONVERT_
          XDR);
        }
      else
        {
          ...
        }
      for(Grid = Start; Grid < Total -1;
      Grid++)
        {
          ...
        }
      if(WhichMachine ! = LAST_GCTM)
        {
      ReadFloat(NextMachineRd, andPrevTimeNext
      Grid, sizeof(float), 1, CONVERT_XDR);
      ...
      WriteFloat(NextMachineWd, andPrevTimePrev
      Grid, sizeof(float), 1, CONVERT_XDR);
      }
    else
      {
        ...
      }
    }
  ...
  for(PipeIndex = 0; PipeIndex < NoOfOpenPipes;
  PipeIndex++)
```

```
    {
      CloseDistPipe(PipeFd[PipeIndex]);
    }
  for(PipeIndex = 0; PipeIndex < NoOfCreate
  Pipes; PipeIndex++)
    {
      DeleteDistPipe(CreatePipes[PipeIndex]);
    }
  CloseGridComputationTask(TaskId);
  return 1;
  }
```

5.6 Performance Analysis

The performance analysis intends to show the speed-up achieved by the parallel execution of the problem and scaled down memory requirements. It presents a case wherein the communication overhead can be concealed to achieve a linear to super-linear speed-up. The analysis discusses the performance resilience of the application, synchronization delay among sub-tasks, effect of the network overhead, and load fluctuations on performance and performance saturation.

The Steady State Equilibrium Problem is considered for our experiments and performance analysis. It represents computationally intensive problems with high memory requirements and patterns in process interactions.

5.6.1 Effect of Memory Scaling

The problem iteratively computes a function on a huge set of grid points. The data required for evaluation of the function, the intermediate data, and the result constitute a large set. Hence, the program may need to use secondary storage during the course of computation. The use of secondary storage during the computation proves expensive in time. When the problem is partitioned, the total memory requirement gets divided among the sub-tasks. In the case we explore, the memory requirement scales down linearly with the number of sub-tasks. This can potentially dispense the expensive I/O during the course of computation. Also, as the memory requirement decreases, performance increases due to cache advantage and reduced memory swapping. The graph in Fig. 5.7 presents a case.

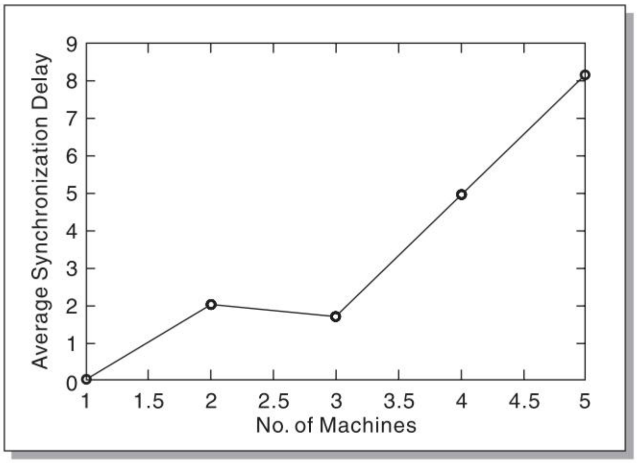


FIG. 5.7

Synchronization Delay vs. Number of Sub-tasks

The graph in Fig. 5.7 quantifies the increase in task finish time with respect to memory requirements. The same program is run for the increasing number of grid points. Consequently, the memory requirement increases. The result shows that task completion time increases more than linearly with an increase in the scale of the problem.

5.6.2 Results of Parallel Execution

The problem was executed parallel-wise on a heterogeneous collection of unevenly loaded workstations. The load variations on individual machines are unpredictable. The problem is run for

1,551 iterations over 100,000 grid points. Each sub-task computes the function over a sub-domain and exchanges its boundary values with the neighbours on either sides once in each iteration. Hence, each sub-task would receive and send data 3,002 times during the course of its execution. The two sub-tasks which compute the two ends of the domain, will have only one neighbour each. The experiment was repeated to parallelize the problem on two to five nodes.

Table 5.1 summarizes the results of parallel execution. The table shows the load on each node (Load), the grain size of computation allotted for each node (Grain Size), the time of completion of individual sub-tasks (Task Time), synchronization delay suffered by each sub-task (Sync. Time), and the speed-up achieved by each parallel run (Speed-up).

Table 5.1 Results with Equal Load Division

#Nodes	*Node no.*	*Load*	*Grain size (Grids)*	*Task time (in sec)*	*Sync. time (in sec)*	*Speed-up*
1	1	10800	200001	348	NA	NA
2	1	14275	100001	161	1	2.148
	2	14358	100000	162	3	
3	1	4570	66667	106	3	3.252
	2	11228	66667	107	1	
	3	9930	66667	106	1	
4	1	17603	50001	86	0	4.046
	2	6841	50000	85	8	
	3	2716	50000	85	9	
	4	3000	50000	85	3	
5	1	4200	40001	69	11	4.971
	2	6508	40000	70	3	
	3	9358	40000	70	6	
	4	2780	40000	69	12	
	5	7358	40000	70	9	

The grain size of a sub-task (Grain Size) represents the number of grid points allotted to the sub-task. Task time of a sub-task is the time taken for completion of the sub-task. It is the sum of the actual CPU time of the sub-task and the synchronization delay suffered by the sub-task. The time of finish of a task is defined as the time at which all sub-tasks of the task are completed. Total synchronization delay of a sub-task (Sync. Time) is defined as the total time the sub-task spent waiting to receive data from neighbouring sub-tasks. It is the sum of the synchronization delay in each iteration. Speed-up is defined as the ratio of sequential execution time of the problem to its parallel execution time.

A larger test on a slightly modified version of the problem yields mostly sub-linear speed-ups due to more time being spent waiting for boundary values from neighbouring nodes. These results are summarized in Table 5.2.

Table 5.2 Results with Larger Tasks

#Nodes	*Grain size (Grids per node)*	*Iterations per node*	*Task time (in sec)*	*Speed up*
1	480000	1000	8722.73	—
4	120000	1000	2423.00	3.60
8	60000	1000	1166.67	7.86
12	40000	1000	781.671	11.22
16	30000	1000	580.71	15.02
20	24000	1000	459.79	18.971

5.6.3 Load Balancing

The load indices shown in Table 5.1 are values obtained from LINUX machines using the *sysinfo* call. The load is balanced over the participating machines by allotting granule sizes of computation according to the current load of the machines. However, the quantitative translation of the load balancing is more involved. The load indices obtained from the system quantify the average contention for the CPU at each CPU time slice. Under lightly loaded conditions, a large number of CPU cycles are wasted. The load index obtained under such conditions ranges from a few hundred to a few thousand. When a CPU intensive process is run on a machine, it contends for the CPU at every time slice.

Correspondingly, on a Linux node, the load index increases by an additive factor of around 60,000. This value is several times more than the values obtained during lightly loaded conditions. When the parallel sub-tasks are started on the nodes, the system works on this high values of load. Due to these observations, we employ the following two heuristics in our load balancing scheme:

- The values of load differences of the order of a few thousand are not substantial and, hence, ignored.
- The load ratio is calculated on the projected values of the load with the load increase that would occur when the sub-tasks start their execution.

The load values obtained from different operating systems are normalized for comparison. Also, a rough index of the processing power of different nodes is derived from the experience of running computationally intensive jobs on them. This index of processing power is also utilized for load balancing.

5.6.4 Synchronization Delay

The total synchronization delay of individual sub-tasks is an important set of parameters for two reasons. It reflects load imbalances and performance saturation. A high value of S_i for one sub-task alone signifies that the sub-task completes its work faster than the neighbouring sub-tasks. Hence, the grain size for the sub-task can be increased to facilitate better load distribution.

As more machines participate in parallel computation, the average grain size decreases. This would result in performance saturation beyond which more machines would not contribute significantly to the speed-up. Performance saturation is reflected as a high value of S_i for most of the sub-tasks.

For each run, more than one node may have a non-zero value for the Total Synchronization Delay. This is explained by spurs of load variations during the run. Consequently, for some iterations, a sub-task may reach the synchronization point ahead of its neighbouring sub-task. Similarly, the neighbouring sub-task may reach the synchronization point earlier during the other iterations.

It can be seen that the Sync. Time is considerably low for the initial experiments. The data required for a sub-task is generated by its neighbour in its last beat of computation. This conceals the communication overhead through the network and achieves linear-like speed-up. The scaled down memory requirements further enhance the performance to achieve super-linear speed-up.

The synchronization delay of a sub-task i in a beat of computation j $(S_{i,j})$ could be expressed as:

$$S_{i,j} = (t_{i,j,available} - t_{i,j,read}) \text{ if } (t_{i,j,available} > t_{i,j,read}), \qquad \text{(equation 2)}$$

$$S_{i,j} = 0 \text{ if } (T_{i,j,available} <= t_{i,j,read})$$

where $t_{i,j,available}$ is the time at which the data is available for sub-task i and $t_{i,j,read}$ the time at which sub-task i issues a read.

The total synchronization delay of a sub-task i (S_i) could be expressed as

$$S_i = \sum_{J=0}^{J=last} S_{i,j} \qquad \text{(equation 3)}$$

The increase in total synchronization delay of individual sub-tasks with respect to the number of sub-tasks in the experiment is shown in Fig. 5.8. The graph shows the average of the total synchronization delay suffered by individual sub-tasks. As the number of sub-tasks for the task increases, the granularity of each sub-task decreases. This potentially results in an increased synchronization delay at each sub-task.

5.6.5 Performance Resilience

It was already mentioned that the data to be read by a sub-task is written by its neighbouring sub-task during its last iteration. Hence, the synchronization delay would become apparent only if the time for the data to be available exceeds the time for the reading sub-task to reach the synchronization point. However, these time durations are probabilistic and are subject to change with load fluctuations of the nodes and network. We define the resilience of performance of a sub-task i in a beat of computation j $(R_{i,j})$ as the sum of delay at the neighbouring task and delay in the underlying network which can be tolerated by sub-task i in the jth iteration without affecting performance.

$$R_{i,j} = \sum_{K=0}^{K=(n_i-1)} tf_k \qquad \text{(equation 4)}$$

where n_i is the number of grids allotted for task i and tf_k the time taken by the task to compute the function f on kth grid. If the sending sub-task and the receiving sub-task were progressing alike, $R_{i,j}$ is the time by which the send would have preceded the receive.

Network Overhead (n_o) is defined as the transit time for a communication. Resilience to Network Load (R_n) is defined as the maximum delay in the network which a sub-task can tolerate without allowing its performance to be affected, when the sub-tasks which send and receive progress alike.

$$n_o = t_{i,\,receive} - t_{i-1},_{send} \qquad \text{(equation 5)}$$

where $t_{i,\,receive}$ is the time at which the receiving sub-task reaches the synchronization point and t_{i-1}; send the time at which the sending sub-task reaches the synchronization point.

The rate of progress of individual sub-tasks may differ due to the difference in load fluctuations of machines. Resilience to load fluctuations during the execution of sub-tasks (Rl) is defined as the difference in time of the completion of sub-tasks that a sub-task can tolerate without allowing its performance to be affected. Load Fluctuation Factor (l_f) is defined as the difference in time of the completion of a beat of computation of the receive task and the send task.

$$l_f = tb_{receive} - tb_{send} \qquad \text{(equation 6)}$$

where $tb_{receive}$ and tb_{send} represent the time taken for a beat of computation by receive and send sub-tasks, respectively. Using (5), (6), and (7), the synchronization delay of a sub-task in a beat of computation ($S_{i,j}$) can be expressed as:

$$S_{i,j} = n_o + l_f - R_{i,j} \text{ if } (n_o + l_f) > R_{i,j} \qquad \text{(equation 7)}$$

$$S_{i,j} = 0 \text{ if } (n_o + l_f) <= R_{i,j}$$

5.6.6 Performance Saturation

The experimental results presented for parallel execution have shown a linear to super-linear scale-up. However, for a given problem, the speed-up does not scale up beyond some value of number of nodes. The granularity of sub-tasks, computational requirements of the function to be evaluated, network overheads, and load fluctuation factor are the factors which affect the saturation point. An increase in the granularity of sub-tasks and computational requirements of the function to be evaluated improves the saturation point of performance. Increases in network overheads and load fluctuation factor result in an early saturation of performance.

Granularity increases if the number of grid points is increased. When the number of nodes increases, the domain is split into more sub-domains. Hence, the average granularity of sub-tasks decreases. This results in an increased synchronization delay at each sub-task. It must be recalled that the time of completion of a sub-task is the sum of its actual CPU time and synchronization time.

$$tt_i = tt_{i,\,cpu} + tt_{i,\,syn} \qquad \text{(equation 8)}$$

where tt_i is the task time of sub-task i, $tt_{i,\,cpu}$ is the CPU time of sub-task i, and $tt_{i,\,syn}$ is the synchronization delay of sub-task i.

The relationship between synchronization delay and granularity was established using (5), (8) and (9). The computational requirements of the function to be evaluated constitute an index of the time taken by the function to compute the value at a grid point. Equation (5) shows its relationship with performance resilience.

The graph in Fig. 5.7 shows a steep increase in synchronization delay as the number of sub-tasks is increased. In the experiment, the performance starts saturating beyond five sub-tasks. We define saturation point as the value of the absolute granularity of sub-tasks at which average synchronization delay of sub-tasks exceeds 10 per cent of the average of task time. However, in practice, a few more nodes could be utilized to effect marginal improvement in speed-up. The absolute granularity of a sub-task is synonymous with its performance resilience. The value of absolute granularity with five sub-tasks is calculated to be around 45 ms. The observed values of round trip network delay among the nodes used for our parallel computation vary from 0.3–3.5 ms with an average of 0.45 ms. This accounts for only 10 per cent of the performance resilience. The difference in load fluctuations accounts for the sum of n_o and l_f exceeding the performance resilience.

5.6.7 Static Overheads

The programming model supports code migration, argument passing, transparent laying of DPs, and result collection. They result in some fixed time overheads. The typical fixed time overheads encountered are given in Table 5.3.

Table 5.3 Static Overheads

Source of overhead	*Order of overhead*
Code Migration Overhead	10s of seconds
Compilation Time	few seconds
Argument Passing Overhead	few seconds
Transparent Pipe Layout Overhead	few seconds

5.7 Future Works

The class of problems with similar patterns in process interactions can be studied. The issues in extending the model for problems of adaptive nature can be explored. The concepts presented in the work can be utilized for adaptive load balancing by load redistribution. The issues encountered while redistributing communicating parallel tasks can be studied. The concepts presented in the work can be integrated into existing high-level parallel platforms to support inter-task communication. The study of many problem domains for the separation of the system and the programmer's concern can lead to generic metaprograms for classes of applications. The system can achieve transparent optimizations in the parallel execution of such metaprograms.

Such metaprograms can also specify the fault tolerance and checkpointing schemes appropriate for them.

References

1. Litzkow, M. and M. Solomon, "Supporting Checkpointing and Process Migration Outside the UNIX Kernel," Proc. Usenix Winter Conf., pp. 283–290, January 1992.
2. Gelernter, D. and D. Kaminsky, "Supercomputing Out of Recycled Garbage: Preliminary Experience with Piranha," Proc. Sixth ACM Int'l Conf. Supercomputing, pp. 417–427, July 1992.
3. Casas, J., R. Konuru, S. Otto, R. Prouty, and J. Walpole, "Adaptive Load Migration Systems for PVM," Proc. Supercomputing, pp. 390–399, November 1994.
4. Joshi, R.K. and D. Janakiram, "Anonymous Remote Computing: A Paradigm for Parallel Programming on Interconnected Workstations," *IEEE Trans. Software Eng.*, Vol. 25, No. 1, pp. 75–90, January 1999.
5. Douglis, F. and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software Practice and Experience*, Vol. 21, No. 8, pp. 757–785, August 1991.
6. Cheriton, D.R., "The V Distributed System," *Comm. ACM*, Vol. 31, No. 3, pp. 314–333, March 1988.
7. Van Dijk, G.J.W. and M.J. Van Gils, "Efficient Process Migration in the EMPS Multiprocessor System," Proc. Sixth Int'l Parallel Processing Symp., pp. 58–66, March 1992.
8. Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors*, Vol. 1. Englewood Cliffs, Prentice-Hall, New Jersey 1988.
9. Sreenivas, T.H., "A Class of Sub-optimal Algorithms for Schedule Optimization Problems," Ph.D thesis, Dept. of

Computer Science and Eng., Indian Institute of Technology Madras, Chennai, July 1998.

10. Hoare, C.A.R., "Communicating Sequential Processes," *Comm. ACM*, Vol. 21, No. 8, pp. 666–677, August 1978.

11. Andrews, G.R., "Paradigms for Process Interaction in Distributed Programs," *ACM Computing Surveys*, Vol. 23, No. 1, pp. 49–89, Engineering, Indian Institute of Technology, March 1991.

Chapter 6

Parallel Programming Model on CORBA*

6.1 Introduction

The Common Object Request Broker Architecture (CORBA) [30] has emerged as a standard for distributed object computing. It is being used for several applications such as building digital libraries [31], scientific problem solving [11], multimedia applications [8], network management [24], etc. This chapter explains the attempt to use CORBA for parallel programming in a Network of Workstations (NOW).

Exploiting NOWs for parallel computing is not easy due to the following factors:

- Load fluctuations in the nodes of the NOW,
- Dynamic availability of the nodes,
- Heterogeneity in architecture and operating systems,
- Varying processor and network speeds, and
- Possibility of failure of nodes.

In the literature, there exist several parallel programming models that address some of the above issues and utilize the NOW for parallel computing. The notable ones include ARC [20], Piranha [7], Mentat [13], the Berkeley NOW project [1], Batrun

*D. Janakiram, A. Vijay Srinivas, P. Manjula Rani

[34], Condor [25], etc. A taxonomy of the programming language level efforts is given in [2].

CORBA is not inherently suitable for parallel programming because of two reasons. First, a notion of concurrency does not exist in the CORBA specification. Second, no services which address any of the above issues have been built over CORBA. P-CORBA brings a notion of concurrency into the CORBA domain. Existing approaches tend to introduce different kinds of objects into CORBA, whereas in P-CORBA, there is only one kind of object that CORBA deals with. P-CORBA also demonstrates how load balancing can be achieved in CORBA-based distributed systems. The load balancing approach of P-CORBA is also different from other existing approaches, in the sense that it builds load balancing as a separate service while approaches such as [3] integrate load balancing into existing CORBA services.

The rest of the chapter is organized as follows. Section 6.2 explains the related works. Sections 6.3 and 6.4 explain the proposed model, with Section 6.3 explaining the notion of concurrency and Section 6.4 explaining the system support that is provided. Section 6.5 explains the implementation and gives the performance studies. Section 6.6 gives a brief summary of the usage of CORBA for parallel programming. Section 6.7 gives future research directions and concludes the chapter.

6.2 Existing Works

To the best of the authors' knowledge, there have been only two other attempts at parallel programming over CORBA. These are the PARDIS [22] and the Cobra [3] systems. PARDIS is a system which ensures that parallel distributed applications interoperate by extending the Interface Definition Language (IDL) of CORBA. An SPMD object is pushed into CORBA to capture data parallel computations. Cobra is an environment for parallel programming in which the main focus is the design of high-performance applications using independent software components. Cobra also introduces a parallel object into CORBA by extending the IDL of

CORBA. The fundamental difference between Cobra and PARDIS is that PARDIS lets the programmer handle object distribution whereas in Cobra, a resource allocator exists which makes the mapping from objects to machines. In both the models, the objects constituting the parallel object communicate through a mechanism outside CORBA, for example a message passing library such as MPI [27].

P-CORBA is similar to the above models in the sense that a new object called a meta-object which looks similar to the parallel object is introduced. The critical difference is that the meta-object is translated by a parser into a CORBA object. The consequence of this is that CORBA can uniformly handle only one kind of object. But in both PARDIS and Cobra, CORBA has to handle two different kinds of objects. This may lead to problems at the middleware level. For instance, the implementation repository which stores address information for activating objects, may have to be extended to handle two different kinds of objects. The entity which handles object location and request delivery[1] needs extension.

The other advantage of the translator-based approach is that the translator and the runtime system can be modified to make use of a different middleware. For example, one may think of using DCOM [9] in some Windows NT clusters since it integrates closely with the operating system and hence performs well.

This work differs from the earlier works in its handling of load fluctuations in a NOW. In Cobra, the resource allocator decides on which machines the objects are to be executed statically at compile time. In PARDIS, the programmer has to make this decision at compile time.[2] Thus, in these models dynamic load fluctuations in the NOW cannot be handled. One of our key observations relating to load fluctuations is that, in a NOW, wide fluctuations in the load of the machines take place during the execution of a parallel program. An important aspect of P-CORBA is that the runtime system is designed to handle this problem. The proposed model can also handle differences in processor speeds through the use of the load factor. The load factor is a function of the processor speed as well as the load conditions on the system.

Further, P-CORBA can handle the dynamic availability of machines. Hence, the most important difference between PCORBA and the other two models is that the latter models do not address issues specific to parallel programming on NOWs.

A component-based architecture for scientific problem solving is presented in [11]. It provides a mechanism by which components built in different languages can be instantiated and run on remote machines. It also provides a resource allocator which lets the user decide on which machine a particular component is to be run. However, the emphasis of this work seems to be more on ensuring interoperability of the components rather than on issues relating to parallel computing.

Charm++ [21] is similar to P-CORBA in that it is also an OO-based parallel programming system. Recent extensions to Charm++ [5] introduce object arrays and dynamic load balancing by object migration. An object array is a multi-dimensional collection of data-driven objects, that allows messages to be sent to any subset of its objects. The object array is similar to P-CORBA's meta-object, as one could form a grid of communicating objects using both. Further, the user does not need to know where the objects reside at runtime as this decision is made by the system dynamically. This is true for both Charm++ as well as P-CORBA. The essential difference between the recent extensions to Charm++ and P-CORBA arises in the actual migration mechanism. In Charm++, after an object migrates, the home processor acts as a location broker. In P-CORBA, the filter acts as a location broker. But in P-CORBA, the filter can throw a *Rebind* exception (CORBA specific user-defined exception) to a client so that subsequent messages do not need to go through the filter [18] whereas in Charm++, every message has to make at least one extra hop [5]. This makes the migration mechanism of P-CORBA more efficient than the mechanism used in Charm++.

The approach closest in resemblance to the proposed model is the Adaptive Runtime System (ARTS) [6]. ARTS is a middleware for parallel processing in the PEACE operating system [35]. It provides runtime services for parallel processing in a distributed object space. These services include naming and binding services as well as services for load management and object migration. The main differences between the proposed model and ARTS are as follows:

- ARTS is not designed to handle heterogeneity (in operating systems or in architectures) whereas P-CORBA can be used over NOWs that could be heterogeneous in architecture, operating systems, etc.
- ARTS has been implemented over a parallel Manna computer, which is a Massively Parallel Processor (MPP) whereas the proposed model has been built over a NOW. Targeting NOWs necessitates the addressing of very specific issues, as this chapter points out.
- ARTS cannot handle inter-task communication. Thus, it cannot model a class of computations such as IGCs whereas, P-CORBA can handle such computations.
- Since ARTS is a new middleware layer that has been built specifically for parallel processing, it could be more efficient. But it may not be a good approach to build a specialized middleware layer for every application. It may, however, also be infeasible to build a general middleware that can handle every conceivable application. Thus, the approach of the authors is better: use the standard middleware for all applications, extending it if required.

P-CORBA is also similar to Mentat [13] in the sense that the underlying philosophy of separation of the concerns of the system and the programmer are the same. Mentat also achieves this separation by relieving the programmer of the task of handling lower level details such as synchronization. However, there are some critical differences. P-CORBA can handle dynamic load fluctuations. However, no concept of object migration exists in Mentat and hence, load balancing is done at compile time. Further, the runtime system in Mentat needs to handle two different kinds of objects, viz. independent and contained objects, whereas in P-CORBA, the runtime system handles only normal CORBA objects. Legion [14] is a middleware approach targeted at providing a global virtual computer abstraction that is built on top of Mentat.

The above-mentioned problem of runtime system handling two different objects also exists in Legion, whereas P-CORBA does not suffer from it. But Legion is targeted at Wide Area Networks (WANs), while P-CORBA currently does not address issues specific to WANs.

6.3 Notion of Concurrency

This section explains the notion of concurrency that is used in P-CORBA. It also gives the application developer's view of the model. The basis of this notion of concurrency is given in the OBS model [19].

The notion of concurrency is based on the sub-contract mechanism which models concurrency at the method level. A method can be invoked concurrently on multiple objects through the sub-contract. The key idea in achieving the sub-contract is the meta-object. The meta-object is an entity which can *aggregate* objects of the same class. Meta-objects can be created by instantiating a meta-class called Parclass. The application developer can *insert* objects into the meta-object and then invoke the sub-contract on the meta-object. This ensures that the method is invoked on all these objects simultaneously.

6.3.1 User Interface

The sample code in Fig. 6.1 is for solving a simple parallel matrix multiplication problem to illustrate the user's view of the model. It shows the syntax of the user program in the model. The matrix class has methods to initialize the matrix, to compute the partial product, and to print the results of the partial computation. Each matrix class computes a portion of the product. In this program, results are not returned to the meta-object for the sake of simplicity. In the main body of the program, the meta-class and objects of the class matrix are instantiated. As shown in the code, the objects can be inserted into the meta-object through the *insert* (symbol <<) operator. After this step, the sub-contract call on the meta-

```
class matrix {
public :
    voild Initialize (..); // Initializes
                             the matrix
    void Inverse (..); // Matrix Inversion
    void PrintMatrix(..); // Prints the
                             matrix
};
main() {
Parclass ParMatrix holds matrix;
// metaclass Parmatrix can hold objects
of class matrix
Parmatrix P; // instantiating metaclass
matrix m1, m2;
        m1. Initialize(..);
        m2. Initialize(..);
P << m1;
    // Insert operator, insert matrix m1
       into metaobject
P << m2;
P..Inverse(U); // subcontract operator,
call for parallel execution of Inverse on
all objects of the metaobject P (U is
unordered)
P..PrintMatrix(0);
// print matrices in order, no concurrency
here (0 is ordered)
```

FIG. 6.1

Sample Code for Explaining the User Interface

object is made through the *subcontract* (symbol..) operator. This ensures that all the objects do the partial computation in parallel.

A runtime system ensures that the objects that are part of a sub-contract can be migrated to the best available machines[3] based on the load conditions (Fig. 6.2). The notable point is that the application developer partitions the data by creating the different objects of the same class and inserting them into the meta-object. The runtime system decides on which machines these objects are to execute at runtime. This concept illustrates how cleanly the functions of the system and the programmer are separated in the model. The meta-object also abstracts the concept of fault tolerance in the sense that if any of the method call fails, it is re-executed on a different node.

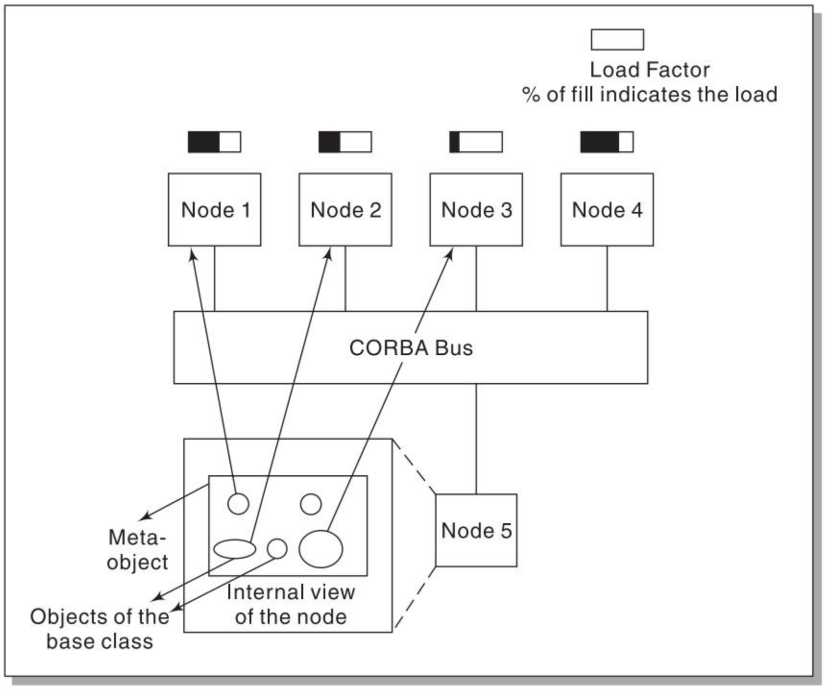

Load Factor
% of fill indicates the load
Node 1
Node 2
Node 3
Node 4
CORBA Bus
Meta-object
Objects of the base class
Internal view of the node
Node 5

FIG. 6.2

An Illustration of the Model

In the OBS model, support for inter-object communication is not provided. So computations involving communicating tasks cannot be modelled. The authors have introduced a new operator called 'message sender' (symbol $0 \rightarrow 0$) on the meta-object through which a message can be sent from one object to another object in the meta-object's collection. The sample code in Fig. 6.3 illustrates how grid computations like the ones discussed in [10] which involve communicating tasks, can be modelled in the program. Each grid object can be initialized with a part of a grid on which it performs the computation. During the computation, it also exchanges boundary values with other objects, say after every iteration. This is done by invoking the message sender operator on the meta-object.

The point to be noted in this context is that in the proposed model, CORBA is transparent to the application developer. From the viewpoint of the application developer, the meta-object and sub-contracting are the key points. The system programmer is the only one who is aware of CORBA. It is the responsibility of the system programmer to ensure that the code written by the application developer is translated into calls on CORBA.[4] This is a fundamental difference between P-CORBA and the other two models, PARDIS and Cobra. In both these models, the application developer is aware of CORBA and the programs on top of an extended CORBA model.

6.4 System Support

This section presents the system programmer's view of the model. The key components of the system include the translator and the runtime system or what the authors refer to as the 'kernel'. The translator converts the application developer's program into calls on the kernel and calls on CORBA's ORB. The kernel is responsible for executing the sub-contract, load balancing and fault tolerance.

```
class grid {
public:
    void Initialize(..); //Initializes the
                           grid
    void Computation(..);//The actual
    computation to be performed
    void PassBoundaryValue(..);//To pass
    boundary value to another object
    void ReceiveBoundaryValue(..);//Receives
    values from another object
};
main() {
grid g1, g2, g3;
Parclass ParGrid holds grid;
ParGrid Pg;// grid meta object
g1. Initialize(..);//similarly for the other
two grids
Pg << g1; //similarly for the other two
grids
Pg..Computation(U);
Pg 0 → 0 g1 to g2;
//message sender operator on the meta object,
message sent from g1 to g2
Pg 0 → 0 g2 to g3;
Pg..Computation(..);
//next iteration
}
```

FIG. 6.3

Sample Code to Explain Inter-object Communication

6.4.1 Translator

The translator parses the program written by the application developer. It converts the meta-object into a CORBA object that aggregates the object references of the objects in its collection. Each object in the collection is converted into a server object in CORBA while the meta-object becomes the client object in CORBA. If the application developer makes the objects in the collection communicate with each other, then these calls are converted into CORBA method invocations by the translator. Hence, the translator is also responsible for ensuring inter-object communication.

This concept illustrates another difference between P-CORBA and the two other models, viz. PARDIS and Cobra. The communication between the objects in the collection is through a mechanism outside CORBA in both the models. But it is not explained clearly how a process-based communication mechanism such as MPI integrates into the object-based communication paradigm of CORBA in these models, whereas in P-CORBA this communication is also through the ORB. However, this may result in a higher overhead for the inter-object communication in the proposed model. But this simplifies the handling of heterogeneity whereas in the other models, the heterogeneity handled by the system is restricted by the mechanism used for inter-object communication.

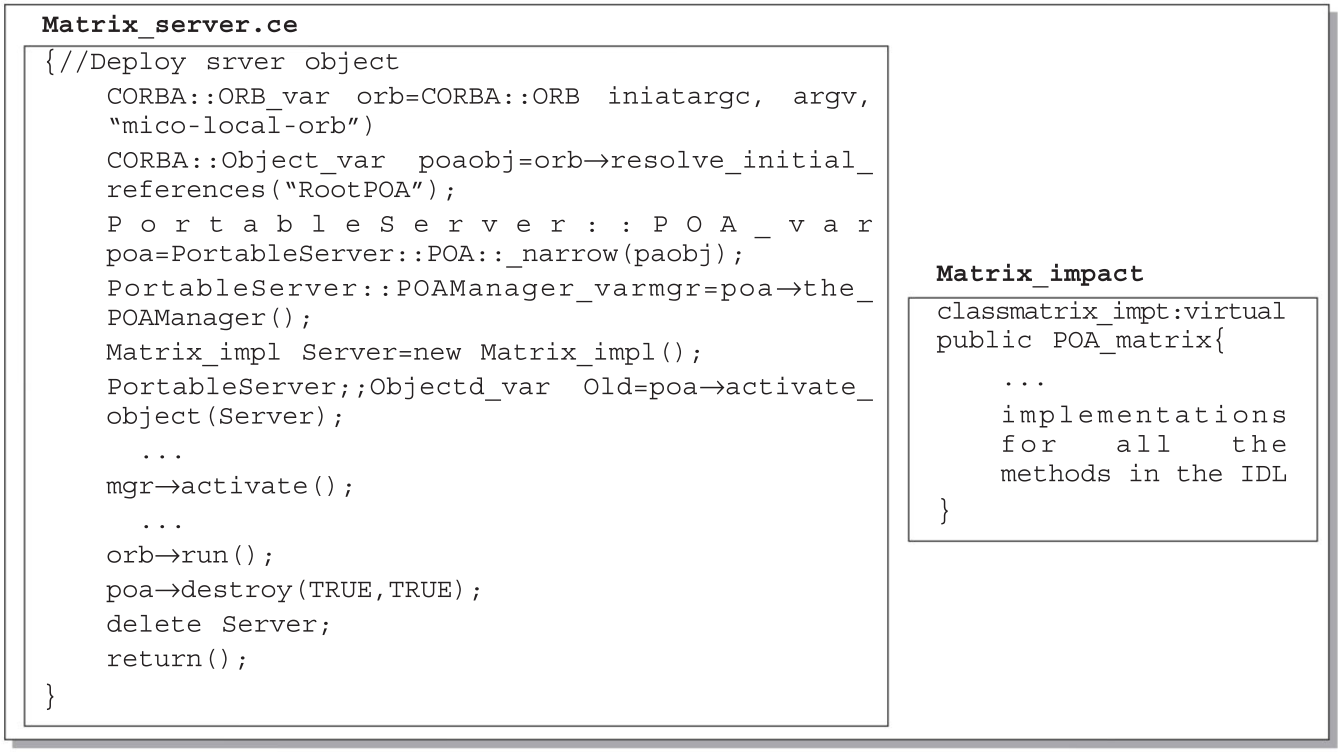
The translator converts the code of Fig. 6.1 into the form shown in Fig. 6.4. This figure shows the IDL file Matrix.idl. It shows some additional methods such as Update, SaveState, etc. These methods are required for the filtering mechanism used during object migration, as will be detailed in Section 6.4.3. The Fig. 6.4 also shows the implementation file Matrix_impl.cc. The Matrix_server.cc file is the code for actually deploying the server objects. The client code is the meta-object, which makes method invocations on all server objects concurrently. The code shown is specific to *mico*, the CORBA ORB used in the implementation of P-CORBA.

Matrix.idl

```
interface matrix {
    oneway void
    Initialize(..)
    void Update(in
    string
    newLocation)
    void Attachi...)
    string
    SaveState(..)
    voidRestoreState(..)
    oneway void
    Inverse(..)
    oneway void
    PrintMatrix(..)
}
```

Matrix_client.ce=MetaObject

```
CORBA::ORB_var  orb=CORBA;ORB_inter(argv,  "nico-
local-orb");
Get object references of server; Can use Naming
Service or Simpler still-mico's bind call.
      CORBA::Object_var  obj1=orb→bind("IDL:
      matrix:1.0", argv(2));
      CORBA::Object_var  obj2=→bind("IDL:
      matrix:1.0", argv(3));
      Corba::Object_var  obj3=orb→bind("IDL:
      matrix:1.0", argv(4));
    Use DLI and make invocation on server object
      matrix_var Client1_ref=SA;;_narrow(obj1);
      CORBA;;Object_ptr  obj_req1=(CORBA;;Object_
      ptr)Client_ref;
    // Initalize matrix
      . . .
    CORBA;;Request  ptr  req1=obj  req1→request)
    "Inverse");
Similarly for the other server object
...
```



Matrix_server.ce

```
{//Deploy srver object
    CORBA::ORB_var orb=CORBA::ORB iniatargc, argv,
    “mico-local-orb”)
    CORBA::Object_var poaobj=orb→resolve_initial_
    references(“RootPOA”);
    P o r t a b l e S e r v e r : : P O A _ v a r
    poa=PortableServer::POA::_narrow(paobj);
    PortableServer::POAManager_varmgr=poa→the_
    POAManager();
    Matrix_impl Server=new Matrix_impl();
    PortableServer;;Objectd_var Old=poa→activate_
    object(Server);
      ...
    mgr→activate();
      ...
    orb→run();
    poa→destroy(TRUE,TRUE);
    delete Server;
    return();
}
```

Matrix_impact

```
classmatrix_impt:virtual
public POA_matrix{
    ...
    implementations
    for all the
    methods in the IDL
}
```

FIG. 6.4 Intermediate Transform Code

6.4.2 The Kernel

The kernel is designed as a distributed kernel that is resident on all the nodes of the system. Each of the kernel entities monitors the load conditions on the respective nodes of the system. When the application developer starts the program on a particular node of the system, the kernel entity on that node is called. This entity interacts with the other kernel entities and gets to know the least loaded machines that are available. It migrates the objects in the program to these nodes and the sub-contract is then executed.

The kernel also takes care of the sub-contract directives and the locking specifications. The kernel uses the concurrency service specification [29] of CORBA for handling locking problems. The concurrency specification provides a mechanism for ensuring the consistency of the state of an object that is accessed by concurrently executing computations. It provides an interface called *LockSet* interface that has methods for acquiring and releasing locks. In the context of the model, the base class is made to inherit from the *LockSet* interface. The kernel acquires the necessary locks before making the method invocation. The concurrency specification provides many locking modes that can be used by the kernel to ensure consistency in the state of the object. But in reality, the authors had to implement the required parts of the concurrency service as most CORBA vendors do not provide an implementation of this service[5].

6.4.2.1 *Load Balancing Strategy*

The most important issue in any load balancing strategy is the load index. Several load indices for measuring the load have been proposed and used. The list includes CPU queue length, time-averaged CPU queue length, available memory and the CPU utilization among others. In the proposed model, the CPU queue length is used as the load index as it has been found to be simple and effective [23]. A threshold policy is used to classify the nodes of the system into three categories. If the load on a node is greater than a threshold, T1, then the node is called a sender. A node is termed as a receiver if its load is less than a threshold, T2, with T2 < T1. Other nodes are grouped into the third category. Nodes in this category do not take part in task transfer. If a user program is initiated at a particular node, then that node automatically becomes a sender.

In sender initiated load balancing algorithms, the task migration is initiated by the heavily loaded node. These algorithms perform well when the load on the system is low, i.e. when it is easier to find a lightly loaded node. When we say that the load on the system is low, it means that the load on a majority of the nodes is low. In contrast, in receiver initiated load balancing algorithms, the task migration is initiated by the lightly loaded node. These algorithms perform well when the load on the system is high, in which case it is easier to find a heavily loaded node. The adaptive algorithm that is used in the proposed model combines the advantages of both the sender-initiated and the receiver-initiated algorithms [33].

There are two major components in the algorithm. One is a sender-initiated component which is triggered on a node when it becomes a sender. The other is a receiver-initiated component which is triggered on a node when it becomes a receiver. It can be observed that a node can become a sender or a receiver at different points of time depending on how the load changes in that node. Each node maintains its view of the load on the system in the form of three lists containing the nodes which fall into each of the categories mentioned above. The information about other nodes is collected when nodes poll each other to initiate the transfer of tasks. The task transfer is equivalent to object migration in the proposed model.

If a node is a sender at a particular time, then it tries to migrate the objects residing in that node to the nodes in its receiver list. If a node is a receiver, then it tries to migrate objects to itself from the nodes in its senders list. Hence, this algorithm performs well when the load on the system is low (or the sender-initiated part dominates) and even when the load on the system is high (or the receiver-initiated component dominates).

6.4.2.2 *Load Balancing Service*

The interface of the kernel object is shown in Fig. 6.5. The load balancing service consists of the collection of kernel objects. The important guidelines that were observed when building the load balancing service are:

- Built on CORBA concepts: The object model of CORBA is strictly adhered to. Thus, concepts like the separation of interface and implementation, and clients depending only on interfaces and not on the implementation are used.
- Allows for local and remote implementations: This could be important, if for instance, the performance requirement of an application is such that the service must be executed in the same process as the client.
- Flexible: Since services are designed as objects, they could be combined in interesting ways. For instance, as shown in this chapter, the load balancing service and object migration service are combined to balance the load on a NOW.
- Finding a service is orthogonal to using it: This means that since services are designed as a collection of CORBA objects, there need not be any special ways of finding them. It is left to the ORB vendor to make the service available to clients. But, this chapter goes a step further and gives general guidelines to the ORB vendor for deployment of the services (refer to Section 6.5.1).

The load balancing service depends on the object migration service for balancing the load on a NOW. This is similar to the dependencies between services as given in the CORBA services specification. For instance, the lifecycle service depends on the naming service and the transaction service depends on the concurrency service.

6.4.3 Object Migration

6.4.3.1 *Introduction*

As the use of distributed object systems is increasing rapidly, it is evident that support for moving objects across machines has

```
Interface Kernel {
    void GetMachinesList(out List L1);
// This method queries other kernel objects, gets load conditions, computes
// load factors and returns list of available machines.
    void GetLocalLoadCondition(out double Bool);
// This method returns the current local load condition. This is called by
// other kernel entities to know the load conditions on this machine.
    void InitiateMigration(string ObjRef);
// This gets list of machines, chooses target for migration, calls migration service
// to migrate the object (with ref ObjRef), to the target machine,
    void QueryForHighLoad();
// This corresponds to the reciever initiated component in the load balancing algorithm.
// This method is called by a receiver that wants to initiate migration. Will check local
// load condition and decide whether to migrate or not.
    void AddMeToMachinesList(string Address);
// This method is called by any new machine joining the system. This is implemented
// as a distributed dissemination algorithm. This is important to handle dynamic
// availability of machines.
}:
```

FIG. 6.5

Kernel Interface

become essential. The following applications may need object migration:

- Large scale distributed systems such as *Globe* [37] for replication and caching purposes,
- Object-based parallel programming [38] for parallelism and load balancing,
- Collaborative applications [39] for information sharing.

Object migration involves two major issues. One is the physical movement of the object with its state. The other is the location problem, which can be posed as, "How can requests sent by clients already holding references to the migrated object be sent to the new location without any changes in the client code or to the object reference?" The physical movement of objects can be difficult to handle if the target object is of a different architecture or if it runs on a different operating system. In such a case, this may require source code migration and recompilation in addition to restoring the state. The location problem could also become difficult, especially if the address of the server is encoded in the object reference.

CORBA [30] is emerging as a standard for distributed object computing. It is being used for different purposes such as parallel programming, building digital libraries, network management, scientific problem solving, multimedia applications, etc. The variety of applications illustrates the need for a proper specification and implementation of object migration in CORBA.

The Life Cycle Service of CORBA [29] describes how clients can control the lifecycle of CORBA objects. It addresses object creation, destruction, copying and movement. The Life Cycle Service implements an operation called *move()*. This method can be invoked by clients for physically moving the object from one machine to another without invalidating the existing references. The *move()* operation gives rise to the following problems:

- CORBA provides location transparency to clients. So, the notion of clients moving server objects is contrary to the object model of CORBA.

- Clients will have to be aware of the protocol used by the target object otherwise, clients may lose connectivity with the target object.
- Existing ORBs are not capable of moving individual objects without invalidating the object reference. This is because the implementation repository cannot store address information at the granularity of objects for scalability reasons [26]. The implementation repository stores entries at the POA (Portable Object Adaptor) level. If only a few objects within a POA move (and not the entire POA), then these objects cannot be located.

The *move()* operation of the Life Cycle Service of CORBA has been described in [15] [16] as 'unimplementable in at least the general case'. Thus, the migration of individual objects is currently a big problem in CORBA.

The fundamental contribution of this chapter is a facility for allowing individual objects to migrate without breaking existing references. The proposed idea bypasses the implementation repository for locating migrated individual objects. Further, the chapter shows how the location mechanism can be extended to a migration service in CORBA.

6.4.3.2 Introduction to Message Filters

This section gives an overview of the concept of message filters and a means of implementing message filters over CORBA. Message filters for object-oriented systems [40] have been proposed to separate message control from message processing in a transparent manner. Message processing refers to the actual actions taken by the server when it receives a client invocation. Message control refers to the intermediate message manipulations that can be done by a filter object. A filter relationship can be introduced between a filter object and its client. Thus method calls to member functions of a server object can be intercepted by corresponding member functions of a filter object. This is possible only if there exists a filter relationship between them[7]. The filter member function can perform the required message manipulations. It can either pass the request to the destination or bounce it back to the source.

The message filters model supports both upward and downward filtering mechanisms. This facilitates the transparent interception of an upward message and its return value. The binding of filter member functions to the corresponding member functions in the client can be dynamic. This means that filters can be plugged or unplugged at runtime. Some instances of manipulations that can be carried out by message filters include transparent security checks, load balancing of replicated servers, maintaining server side caches, etc.

6.4.3.2.1 *An Implementation of Message Filters Over* CORBA

A mechanism for implementing message filters over CORBA is given in [41]. A special interface called 'filterable interface' which has two methods, *attach()* and *detach()*, is provided. These methods can be used for dynamically plugging and unplugging a filter object. A server object that needs to be filtered must inherit from this interface at the IDL level. An implementation of the server inherits from a library implementation of the filterable interface, called *Filterable_impl*[8]. The IDL compilation phase generates a filter base interface as an IDL for implementing filter objects. The methods in the filter base interface are invoked automatically when the corresponding methods in the server are invoked by clients.

A filter object can be implemented by providing an implementation of the filter base interface or by implementing a filter interface that inherits from the filter base interface. This method of implementing message filters requires two main modifications to the IDL compiler. One is to generate the filter base interface. The other modification involves the *dispatch()* method of the server skeleton. This method is modified to invoke the member function in the filter before the corresponding member function in the object is invoked. In a similar way, by modifying the respective methods in the stubs, it is possible to have filters in the client side to filter the return values.

6.4.3.3 *Locating Migrated Objects: A Filter-based Approach*

In this section, it is assumed that an external entity (for instance, a migration service) physically moves the object from one location to the other[9]. This section explains the use of message filters for locating migrated objects in CORBA. The key idea for locating a migrated object is the use of message filters. Every object has a filter plugged to it, at its home location. If the object migrates, the filter (which is always static) is updated with the new location. Thus, this filter acts as a location broker for the object. This is illustrated in Fig. 6.6.

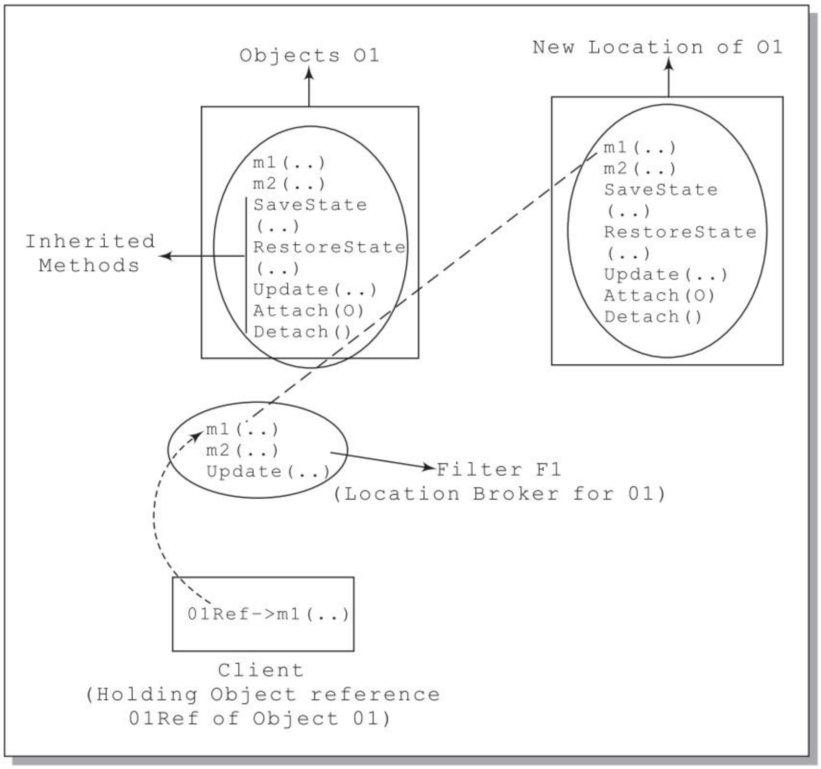


FIG. 6.6

Filter as Location Broker

An object which should be migratable must inherit from a standard interface called *migratable* interface (refer to Fig. 6.7). This interface has methods *SaveState()* and *RestoreState()*. These are used by the migrating entity to save and restore the object state during migration. The other method of this interface is the *update()* method. This is used by the migrating entity to inform the filter about the new location of the object. The migratable interface itself inherits from the filterable interface. This means that if an object needs to be migrated, it implicitly gives permission for filtering. The server implementation must inherit from a library implementation of the migratable interface called *Migratable_impl*[10]. The *Migratable_impl* inherits from the *Filterable_impl* so that implementations of the *attach()* and *detach()* methods are available.

```
  interface Migratable: Filterable{
string SaveState();
void RestoreState(in string State);
void update(in string NewAddress);
}
```

FIG. 6.7

IDL of the Migratable Interface

The inheritance hierarchy mentioned above has the following two important implications:

- It facilitates dynamic plugging (and unplugging) of filters. A migrating entity can plug the filter to the object by calling the *attach()* method on the object reference.
- It provides an elegant updation mechanism. By filtering the *update()* method on the object, the filter can update the changed location of the object.

The redirection of the client requests by the filter methods can be modelled in two ways in CORBA. These are explained in the next two sections.

6.4.3.3.1 *Filter as Client*

The filter method is always called before the call to the server. This method acts as a client to the object in the new location. The following two requirements need to be met if this is to be achieved in CORBA:

- The filter must acquire a reference to that object. This can be acquired by the filter in certain specific ways. For instance, in our current implementation using the *mico* ORB [42], the filter makes a *bind()* call to the new location.
- For invoking the method, the filter should be able to marshal the parameters of the call. This is also possible because the stub is attached to the filter when the IDL file is compiled.

The filter gets back the result of the invocation and subsequently returns the result to the client. This means that with respect to the original semantics of message filters, the filter does a 'bounce' on the request and not a 'pass'. Thus, transparent redirection of client requests is achieved.

6.4.3.3.2 *Filter as Forwarder*

An ideal approach to model the filter as forwarder is as follows. The filter gives a 'location forward' reply to the client with the new address of the object. The client gets the new location and rebinds to that address. As a result, it gets a new reference to the object. However, there are two problems associated with this scenario, which are:

- The filter cannot give a 'location forward'. As per the POA specification of CORBA, only the servant locator or servant activator entities can throw the ForwardRequest exception. This exception is propagated back to the client as a LocationForward reply. The client side ORB can reconnect to the new location.
- If the filter throws a user-defined exception, this is propagated back to the client. But this makes the exception visible to the client. Thus, it violates client transparency.

The above-mentioned problems can be handled by using a client side filter. The original filter (from now, on this is referred to as the server side filter) throws a non-system exception called *Rebind* exception. This exception is caught by the client side filter. The new location of the object is transmitted as a string in the exception body. Thus, the client side filter rebinds to this new location and gets the new reference. The client side filter marshals the request by using the same stub that was used by the client initially. It makes the method call on the reference and gets back the result. It subsequently returns the result to the client. For implementing this approach, the following two modifications are necessary.

- The *Rebind* exception must be understandable to both the client and the server. It is declared in the IDL of the *Migratable* interface[11].
- The stub must be modified to implement the client side filter. This is similar to the modification of the stub to implement server side filters.

A comparison of the two approaches is given in Table 6.1.

Table 6.1 A Comparison of the Two Approaches

Filter as forwarder	*Filter as client*
Processing is negligible	Filter marshals parameters
No extra delay, even for large message sizes	Extra time delay for large message sizes
Can handle simultaneous requests and large message sizes	May not be able to handle simultaneous requests and large message sizes
Needs additional modifications to the IDL complier	No such modifications are required

6.4.3.4 *Migration Service in CORBA*

This section explains the extension of the location mechanism to a migration service in CORBA. In the first part of the section, the functions of the migrating entity are described. The modelling of this entity as a CORBA object and of its function as a migration service, are explained in the next part.

6.4.3.4.1 The Migrating Entity

The migrating entity is responsible for:

- The physical movement of the object across machines,
- Plugging the server side filter,
- Handling the 'window' period during object migration.

This entity can be called by a load balancing entity for object migration, or a system administrator for administrative reasons or by the application itself. Following is the sequence of steps that should be followed by this entity when called upon to migrate an object:

- It invokes the *attach()* method on the object reference. This ensures that the filter is plugged to the server object, at its home location. If the migrating entity is never called to migrate a server object, then no filter is plugged to this object. This means that objects which are static will not have the filter plugged. Thus, method calls go directly to the object (unfiltered). Once the filter is plugged, it receives all method calls that are made on this object.
- Then it invokes a method called *SaveState()* on the object. The object saves its state in a file and returns a unique file name[12] to the migrating entity.
- The migrating entity migrates the source code of the object to the new location, similar to code mobility [43].
- It creates a new object at the new location. This is possible once the source code is migrated and recompiled. In CORBA, this means that a new server object of the same type has been created at the new location.
- The migrating entity migrates the state of the object to the new location. It sends a *RestoreState()* method to the new object[13]. This restores the state of the object. The new object is now ready to receive client requests.

- The migrating entity sends the *Update()* method on the original object reference. This call is trapped by the server side filter (at the home location of the object). The filter updates its view of the location of the object.
- The migrating entity does not advertise the new object reference. This means that clients always get the original object reference. Client requests always come through the server side filter which re-directs it to the new location.

The migrating entity takes care of the 'window' period during object migration. The filter traps *SaveState()* method call and sets an internal flag called ObjectInMotion. The *Update()* method, which is also filtered, resets the flag. Thus the flag is set during the window period. Client requests arriving during this period are sent back with a 'Transient' exception. From the client's perspective, this means that the object is currently not reachable. The client could retry after some time. But a clever solution is for the client side filter to trap this exception. The client side filter could re-try after a given time-out period. It could also propagate the exception back to the client code.

6.4.3.4.2 A Migration Service

The migrating entity is modelled as a CORBA object that has a method called *migrate()*. This method takes as parameter a string. This is the stringified form of the reference of the object to be migrated. The other parameters of this method call include the (current) location of the object and the target location. The code for the *migrate()* method based on the sequence of steps outlined above is shown in Fig. 6.8. A prototypical implementation of the migration service is in the testing phase.

The ORB vendor must address two issues with respect to the migration service. One is the method by which applications acquire an object reference of the migrating entity. The other is the deployment of the migrating entity. These two issues are discussed below.

```
class MigrationService_impl : public.. {
..
    void  Migrate  (string  Ref,  string
    OldAddress, string NewAddress) {
      objREf->attach (Filter);
      // object reference obtained from Ref
      StateFile = objRef->SaveState ();
      move  source  code  of  object  from
      OldAddress to NewAddress.
      move  StateFile  from  OldAddress  to
      NewAddress.
      create  new  object  at  NewAddress  with
      NewRef as reference.
      NewRef->RestoreState (StateFile);
      ObjRef->Update (NewAddress);
    }
```

FIG. 6.8

Migration Service Implementation

There are two ways whereby the object reference of the migrating entity can be made available to applications. These are:

- Advertise the object reference of the migrating entity in naming/trading services. This method is easier to implement as it requires no modification to the ORB.
- Modify the *resolve_initial_references()* method of the ORB. In this case, the migrating entity is included in the list of basic services which can be resolved by the ORB itself. This is more difficult to implement as it necessitates modification of a method of the ORB.

There are two ways of deploying the migrating entity. These are:

- There can be one migrating entity for every machine of the system. This method reduces the time required to migrate an object. This is because two of the method invocations made by the migrating entity [*SaveState*() and *RestoreState*()] become local method calls.
- A group of machines can use a single migrating entity. This method may be easier to implement, especially if the machines have a shared file system. But in this case, there is a limit on the number of objects that can be handled by this single entity.

The above two issues should not be considered in isolation by the ORB vendor. It is better to modify the *resolve_initial_references()* method of the ORB if the deployment policy is one entity for every machine. The method can return a local pointer instead of an actual object reference. But if the other deployment policy[14] is used, then it is better to advertise the object reference of the migrating entity.

Different applications can make use of this migration service to migrate objects. A specific policy for selecting the target nodes for migration could be built into the migration service. The migration service could also work in tandem with other services such as a load balancing service [3] to determine the target for migration. The other way to determine the target location is for the application programmer to specify it on the basis of some application-specific criteria.

6.4.3.5 *Implementation*

The mechanism for locating a migrated object by using message filters has been implemented over a 10 Base T ethernet network, with all the nodes running the LINUX operating system. The CORBA implementation that was used is a public domain ORB called *mico*. It is a fully CORBA-compliant ORB with a C++ language mapping. This section describes the implementation of the location mechanism using message filters. Another approach for locating migrated objects by using a servant manager is also explained. The performance-related aspects are discussed.

6.4.3.5.1 *Filter as Client*

Figure 6.9 shows the sequence of steps that are required to make the client requests reach the migrated object. The method call made by the client is intercepted by the server side filter. This filter makes a bind call to the current location of the object. This returns the new object reference to the filter. The filter subsequently makes the method invocation on this reference. As per the semantics of message filters, the filter bounces the request with the return values of this method invocation. The code for the filter implementation is illustrated in Fig. 6.10.

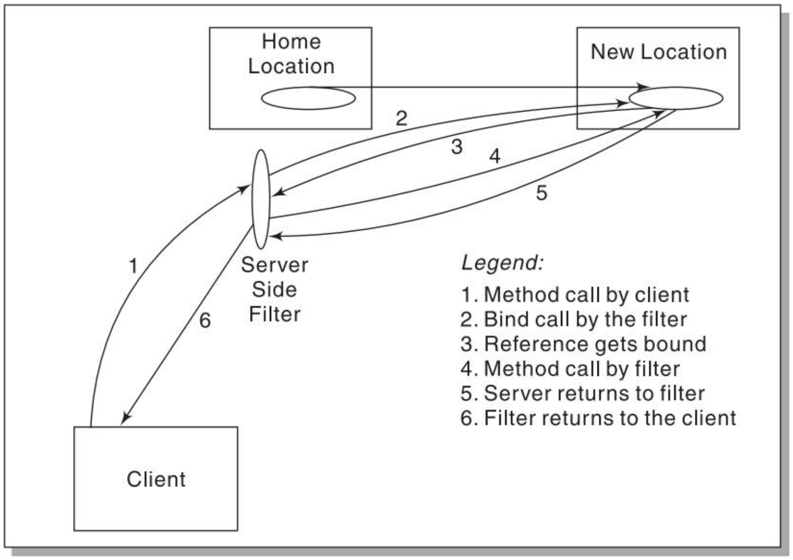


Fig. 6.9

Filter as Client

6.4.3.5.2 *Filter as Forwarder*

Figure 6.11 shows the sequence of steps in this case. The method invocation is intercepted by the server side filter. The server side filter throws the *Rebind* exception. This exception is caught by the

```
class FilterAsClient : public ... {
private:
..
public:
    ReturnType m1 (..){
      // filtering method m1 () in the
      object.
      Address = LookUp();
      // lookup for current object address,
      may be from a persistent store (this
      may be required if object is
      persistent).
      ref = orb→bind(Address,ObjectName);
      // bind to new location and get
      reference
      Objref = InterfaceName ::
      _narrow(ref);
      // narrow to appropriate type
      ReturnValue = ObjRef→m1 (..);
      // actual method call.
      return ReturnValue;
      // return to client.
    }
};
```

FIG. 6.10

Code: Filter a Client

client side filter. The client side filter makes the bind call to the new location. It gets the new object reference and makes the method invocation. Finally, it returns the results to the original

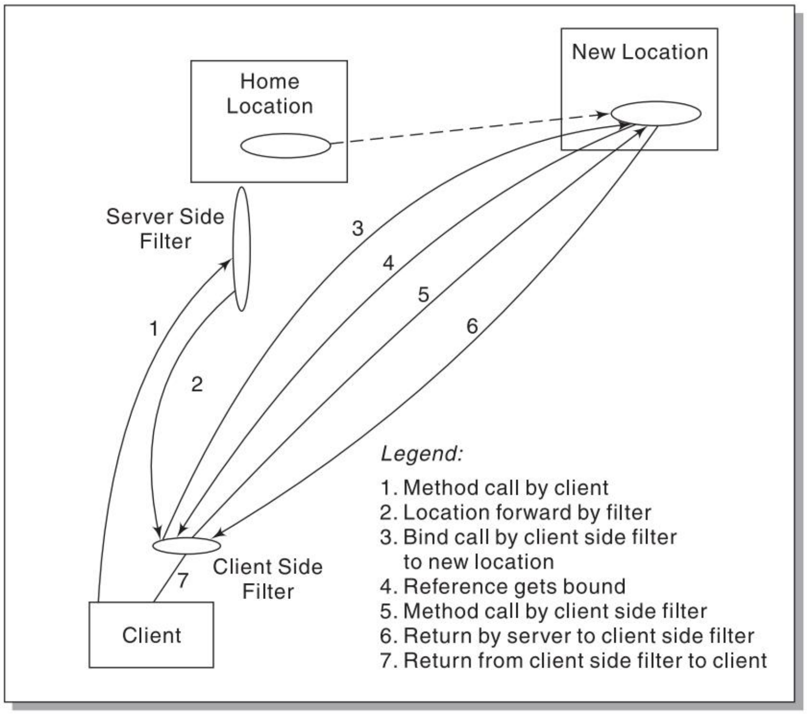

New Location
Home Location
Server Side Filter
Client Side Filter
Client
1
2
3
4
5
6
7
Legend:
1. Method call by client
2. Location forward by filter
3. Bind call by client side filter to new location
4. Reference gets bound
5. Method call by client side filter
6. Return by server to client side filter
7. Return from client side filter to client

FIG. 6.11

Filter as Forwarder

client. The code for both client side and server side filter is shown in Fig. 6.12.

6.4.3.5.3 *Using POAs Servant Manager*

The specification of the POA in the current version of CORBA explains how the POA can be configured to use a servant manager. The servant manager provides implementations for CORBA objects (servants). The servant manager can be of two types depending on the ServantRetention policy of the POA. The first type must support the ServantLocator interface while the other must support the ServantActivator interface [16]. The implementations of both the ServantLocator and ServantActivator can raise

```
class FilterAsForwarder: public ...{
// Server side filter
..
public:
      ReturnType m1 (..){
        Address = LookUp();
        // same as before.
        throw Rebind(Address);
    }
};
class ClientSideFilter : public ...{
// client side filter implemented as a proxy.
..
public:
    ReturnType m1 (..){
      try {
      ReturnValue = OldServerReference→m1 (..)
      return ReturnValue;
    }
catch(Rebind r) {
      ref = orb→bind(r.Address, ObjectName);
      // bind to new location and get reference
      ObjRef = InterfaceName::_narrow(ref);
      // narrow to appropriate type
      ReturnValue = ObjRef→m1 (..);
      // actual method call.
      return ReturnValue;
      // return to client.
    }
};
```

FIG. 6.12

Code: Filter as Forwarder

a ForwardRequest exception. If the ORB uses Internet Interoperability Protocol (IIOP), then this exception is propagated back to the client as a Location_Forward reply. The body of the exception contains the new object reference. It is the responsibility of the client side ORB to reconnect to the new location.

There are two main problems in using this method to locate migrated objects in CORBA. First, if the object moves further, then there must be a servant manager at each of the old locations of the object. This method cannot be modelled as a home-based model. This is because updation is not easy[15]. This makes the approach a chain-based one for locating migrated objects. The performance of the chain-based model may degrade if the chain length increases. This is further illustrated in the performance studies.

Another difficulty of using this scheme arises because the servant manager is at the POA level. The identity associated with objects within the POA called 'oid' must be generated by the application. The oid must be passed to the *incarnate()* of ServantActivator or *preinvoke()* of ServantLocator. These methods can return different locations to the clients, based on the oid (this may be the case, since different objects could move to different locations). The application programmer must be aware of all these details to locate migrated objects, whereas in the case of the filter based model, the migrating entity takes care of these details. Further, since the filter code is the same for all objects[16], it can be generated automatically.

6.4.3.5.4 *Performance Studies*

Two experiments have been conducted on the basis of the three implementations described above. The goal of the first experiment is to show that the home-based model (the filter approach) performs better than the servant manager-based chain model. If an object moves from its home location to another machine, it is said to have migrated by one hop. The experiment was conducted with the object moving an increasing number of hops.

The performance of the servant manager approach degrades as the number of hops increases. The filter approach (for this experiment, the model used was the filter acting as client), however, performs well irrespective of the number of times the object moved. This is because the filter always acts as a home agent and connects directly to the new location of the object. The result of this experiment is shown in Table 6.2.

Table 6.2 Comparison of Filter and Servant Manager Approach

Number of hops	*Filter as client*	*Servant manager as forwarder*
	Time in milliseconds	
1	18.45	13.78
2	21.75	21.70
3	21.75	32.30
4	21.75	36.89
5	21.75	44.25
6	21.75	51.05

The other experiment that was conducted in entailed comparisons of the two variants of the filter-based approach. The performance of the two approaches was measured with increasing message sizes. As the message size increased, the filter as client approach started performing poorly, as compared to the filter as forwarder approach. The reason for this is that in the filter as client approach, the filter has to marshal the parameters again. If the message size is large, the time for marshalling the parameters increases. Hence, the result is as shown in Table 6.3.

Table 6.3 Comparison of Two Variants of the Filter Approach

Method parameter	*Filter as forwarder*	*Filter as client*
No Parameter	18.65 milli secs	18.45 milli secs
Float	20 milli secs	22 milli secs
2-d Array of size 500*500	600 milli secs	1.3 secs
Structure of 2 arrays and 3 long integers	2.5 secs	4.5 secs

6.4.3.6 *Related Work*

Object migration in distributed object systems has been addressed in several systems such as Emerald [44], Mobile Network Objects (MNO) [45], Shadows [46], etc. In this section, the filter-based migration model is compared with these models.

Emerald was among the earliest distributed object systems to provide support for object mobility. It uses the concept of object descriptor which consists of a forwarding address and an object identity (oid). It provides support for different granularities of migration. This means that objects from small data objects to large process objects can migrate. Each node maintains information on local objects for which remote references exist and on remote objects for which local references exist. Further, locating an object involves tracing through the path of forward references. Sometimes, even broadcasting may be involved. For these two reasons, the migration mechanism of Emerald is not a scalable one. However, the filter-based approach is scalable. Since a per-object filter is used, there are no centralized components that impede scalability.

MNO is the work that is closest to the filter-based approach. MNO uses object wrappers called 'proxies'. The proxy executes some code just before and after the invocation on the object. This makes the proxy of MNO similar to the filters used in this chapter. But there are some critical differences. First, the client must get a reference to a proxy and not to a server object in the MNO. But in the filter approach, clients get reference to only the server object. The filter is transparent. Further, plugging a filter can be dynamic. The other difference between MNO and the filter approach is that MNO uses a chain-based forwarding model whereas the filter uses a home-based model and this has been shown (in Section 6.5) to perform better than a chain-based model.

The Shadows system supports mobile objects through the use of a location-transparent object reference scheme. Objects can migrate between object managers which are entities in each node that manage objects within that node. The main contribution of the Shadows system is that it incorporates fault tolerance mechanisms that allow for both safety and liveness of objects. But the concept of compound references that it uses for redundancy could complicate the invocation path and result in a performance penalty. Further, it uses a chain-based forwarding model, which has been shown to perform poorly as compared to the home-based approach of this chapter. However, the filter-based approach does not address fault tolerance currently.

Interceptors were proposed in [47] and subsequently included in the CORBA specification. Interceptors are similar to the concept of message filters used in this chapter. Hence, one can think of solving the object migration problem by using interceptors. There are a couple of problems in this approach. First, to the best knowledge of the authors, the idea of using interceptors for object migration has not been explored in the literature. Secondly, there are some important differences between message filters and interceptors. These are:

- The main motivation for proposing interceptors was fault tolerance and security, not concerns such as replication or migration.
- Message filters can be plugged at runtime. This is not possible with interceptors. Thus, method calls to a static server object will not be filtered. But they will be intercepted (if interceptors are used).

6.4.3.7 Summary

This section has presented a simple and elegant mechanism for locating migrated objects in CORBA by using message filters. The location mechanism has been extended to an object migration service in CORBA. The location mechanism has been implemented while the migration service is in the prototypical stage. The contribution of the model becomes significant because it allows individual objects to migrate without involving the implementation repository. Currently no mechanisms exist in CORBA that allow individual objects to migrate without compromising on the scalability of the ORB. Further, the message filter-based location mechanism is scalable.

The idea of using message filters for locating migrated objects can be extended to other middleware such as DCOM (Distributed

Component Model). This could result in message filters becoming a generic and scalable solution to the location problem in distributed object systems. Methods for incorporating fault tolerance into the location mechanism can also be explored.

6.4.4 Method Invocation

After the objects are migrated to the best possible machines in the system, the meta-object (now the client) invokes the method on all the server objects simultaneously. One way of achieving concurrency at the client side is by using the one-way calls of CORBA's Dynamic Invocation Interface (DII). But this does not apply to the proposed model because the client has to receive the replies from the servers to collect the results of the computation. So the only option left is to use the deferred synchronous calls of the DII. This requires the ORB to be multi-threaded. But few non-commercial ORBs provide efficient support for multi-threading. Thus, if a client makes a number of parallel calls, it has proved to be difficult to implement.

Support for this also comes from the Asynchronous Method Invocation (AMI) which is a messaging specification [28]. By using the AMI, a client can make a request on multiple server objects concurrently and later receive the results by polling or by a call-back model. But, the problem is that the AMI has only recently been drafted into the CORBA specification and only few, if any, ORBs provide implementations of the AMI.

6.5 Implementation

P-CORBA is being implemented on a network of workstations consisting of an IBM Intellistation and three Pentium machines of different vendors connected through a 10 base-T ethernet, all running LINUX-operating system. The implementation of CORBA that is being used is *mico* [32], a public domain implementation. It is fully compliant with CORBA and has an IDL mapping to C++. This section gives an overview of the implementation.

The main entities of the implementation are the kernel and the translator. The kernel is modelled as an active object that is instantiated on all the nodes of the system. It monitors the load conditions on that machine by using the *uptime* call to the LINUX kernel. It also maintains its local view of the load conditions on the other nodes of the system. This view is updated whenever any communication is received from any of the other kernel objects. This is one of the ways of maintaining information in a load balancing algorithm as suggested in [33].

The translator parses the user program and converts the objects into server objects in CORBA. This is done by first generating an IDL file, generating the implementation[17] and finally generating the server code. The translator also converts the meta-object in the user's program into a CORBA client object. The object references of the server are stringified and are made data members of the client object. The translator finally adds a method called *subcontract*() to the client object.

The client object interacts with the kernel object of that node to specify which objects in its collection are to be migrated. The kernel object decides on which machines these objects are to be migrated and performs the actual migration. This ensures that the computation is started off on the best possible nodes. The sub-contract is then executed. If any of the calls fail, then that call is re-executed on a different node. Figure 6.13 shows the interfaces of the meta-object (aggregate CORBA object).

The important point to be noted is that since the kernel object is a CORBA object, it can be accessed through the ORB. If the resolve_initial_references() method in the CORBA ORB can be modified to give a reference to the kernel object, then this could become a stand-alone load balancing service in CORBA. The list_initial_services() method also needs to be modified if it is implemented by the particular ORB. This has been implemented in the mico source code and is currently being tested. This approach is different from the one suggested in [3] for load balancing in CORBA-based systems. In this approach, the load balancing functionality is integrated into an existing service, the naming service of CORBA. The naming service chooses one of several replicated servers based on load conditions, when a client

```
// the meta-object is only an aggregate
COBRA object.
meta-object {
ObjRefList; // List of object references
part of the meta-object.
also contains the list of source codes and
states.
Subcontract (..)
// Called by the translator entity within
the node. handles the subcontract.
}
```

FIG. 6.13

Interface of the Meta-object

makes a request for name binding. The naming service returns a transient object reference, forcing the client to rebind through the naming service after every session.

6.5.1 Deployment of Load Balancing Service

The ORB vendor must address two issues for providing any service to clients. One is the method by which applications acquire an object reference of the service. The other is the deployment of the service. These two issues are discussed below.

There are two ways by which the object reference of the service can be made available to applications. These are:

- Advertise the object reference of the service in naming/trading services. This method is easier to implement as it requires no modification to the ORB.
- Modify the *resolve_initial_references()* method of the ORB. In this case, the service is included in the list of basic services which can be resolved by the ORB itself. This is more difficult to implement as it requires modifying a method of the ORB.

There are two ways of deploying the service. These are:

- The objects that form the service can reside on every machine of the system.
- A group of machines can be configured to share the service.

The above two issues should not be considered in isolation by the ORB vendor. It is better to modify the *resolve_initial_references()* method of the ORB if the deployment policy is one entity for each machine. The method can return a local pointer instead of an actual object reference. But if a number of machines are configured to share the service, then it is better to advertise the object reference of the service.

In the case of the load balancing service, the deployment policy must be one kernel entity per machine (to monitor the load on every machine). Thus, it is better to modify the *resolve_initial_references()* method of the ORB. This method can be made to return a local pointer as a result. This optimization can be done by the ORB vendor.

6.6 Performance

This section details the performance studies conducted over a prototypical implementation of P-CORBA that has been built and tested. A comparison with MPI, a widely used parallel programming tool, is also presented. The first case study was done over an implementation of the Genetic Clustering Algorithm (GCA) [17] for the TSP. The second case study uses an IGC problem.

6.6.1 Case Study 1

The key concept behind GCA is that it is a combination of Simulated Annealing (SA) and Genetic Algorithm (GA). SA is a neighbourhood algorithm in the sense that given a good initial solution, it can converge to the actual solution. GA is an algorithm adapted from the field of genetic engineering for improving the whole population. This means that GA can search the whole solution space and potentially yield a number of good solutions.

Hence, the GCA uses the GA for generating the initial solutions to be fed to SA. The main reason why this problem was chosen is that the computation to communication ratio is high. This kind of coarse-grained problems are well-suited to run on NOWs.

In the context of the model, the client runs GA and gets the required number of good initial solutions. The servers run SA, with each one searching a particular region of the solution space. A similar prototype was implemented over MPI also. The client in this case is the native process which sends the GA results to all the other processes. These processes run SA.

Table 6.4 shows the overhead of using MPI versus the overhead of using CORBA. This was measured by making a simple method call in the case of CORBA. In the case of MPI, the overhead was measured by a simple send call. These measurements were made on a 10 Mbps ethernet cluster. Table 6.5 shows similar measurements over a 100 Mpbs ethernet cluster.

Table 6.4 Comparison of Overheads of CORBA and MPI: 10 Mbps Network

Data sent	*Method call in CORBA*	*Message send in MPI*
	(Time in milliseconds)	
No data	0.996	0.169
Integer	1.0	0.276
Float	1.2	0.327
Long array of 500	890	82.9

Table 6.5 Comparison of Overheads of CORBA and MPI: 100 Mbps Network

Data sent	*Method call in CORBA*	*Message send in MPI*
	(Time in milliseconds)	
No data	0.458	0.15
Integer	0.519	0.198
Float	0.533	0.255
Long array of 500	111	14

Table 6.6 shows the worst case, average case and the best case time delays for the GCA over P-CORBA and MPI. The best case delay was measured when load on the system was very less (almost no load). The worst case delay was measured when the load on the nodes (on which computation is started) becomes high. The average delay was measured by evaluating the performance over different load conditions. The load was measured by using a Linux specific system call *sysinfo*. The performance studies were conducted on a cluster of eight machines, with the computation being started on four of the eight machines. In the heavily loaded case, six of the machines are loaded, while two machines which did not start the computation were free. In the lightly loaded case, all the eight machines are lightly loaded. In the average case, two machines of each category (which started the computation and ones which did not start the computation) were heavily loaded. The machines were artificially loaded by a *CPUhog* process[18].

Table 6.6 Performance of CORBA and MPI for GCA

Load conditions	*CORBA*	*MPI*
(on nodes in which computation starts)	*(Time in seconds)*	
Light (best-case)	35	34
Average	39	45
Heavy (worst-case)	45	59

It can be seen that under light load, MPI and P-CORBA perform nearly the same, with MPI having a slight improvement. However, under average (or heavy) load conditions, MPI performs poorly. But since P-CORBA can handle dynamic load fluctuations, it performs well under these conditions.

The point to be noted from the results is that the overhead of using MPI is much less as compared with that of using CORBA. But P-CORBA clearly outperforms MPI. This is because PCORBA ensures that computation does not continue on a heavily loaded machine whereas in MPI, the task continues to run on the same heavily loaded node and incurs significant overheads. Thus, in MPI, even though computation can start off on the best possible nodes, dynamic load fluctuations cannot be handled. The set of machines on which computation starts is the same as the set in which computation ends. But in P-CORBA, the two sets can be different which results in better load adaptability and hence, in increased speed-up.

6.6.2 Case Study 2

The main motivation for the second case study is to illustrate the modelling of communicating tasks. The prototype for this case study consists of an implementation of the *steady state equilibrium problem*. The problem is to compute the temperature distribution of a rod whose ends are kept at fixed temperature baths. This problem is taken from the area of fluid dynamics [4]. It falls in the category of IGCs, which comprises a large class of engineering applications.

The problem iteratively computes the temperature values at equally spaced grid points, at regular intervals of time. Each iteration consists of computing the function for a time slice. The temperature of a grid at a particular time slice is a function of the temperature of this grid and its adjacent grids in the previous time slice. This is depicted in Equation 1. This dependency arises due to the influence of conduction in the temperature distribution. The problem considers the flux in only one dimension.

$$T_{x,t} = f(T_{x,\,t-1},\ T_{x-1,\,t-1},\ T_{x+1,\,t-1}) \qquad \text{(equation 1)}$$

where $T_{x,\,t}$ refers to the temperature at a grid point x at time t.

The experimental set up was a cluster of Linux-based Intel PCs, with 256 MB of main memory and 333 MHz Intel Pentium PII processors, connected by a 10 Mbps LAN. Table 6.7 shows the speed-up obtained as a result of parallel execution of the IGC problem. The grain size is the number of grids that is allocated to each computing entity. The task time is the time taken for the entity to calculate the temperature distribution over the region allocated to it, for a fixed number of iterations (till a steady state is reached). For instance, the task time in Table 6.7 was computed after running the problem for 1500 iterations. The synchronization time is the time taken by each entity to obtain the data for its next iteration. The synchronization time includes the overhead of one object migration. One of the machines was artificially loaded by the CPUhog process. When the load increased beyond a pre-defined threshold, the task was migrated to a different node (lightly loaded).

In Table 6.7, the speed-up obtained is linear up to five machines. However, for six machines, the speed up is sub-linear because the problem has reached saturation point (this is evident from the fact that the synchronization delay for all the nodes is very high). This means that the time taken for one iteration (by any of the computing entities) is nearly equal to the communication delay (message sending overhead) between the entities. If the computation time is lowered any further (by reducing the grain size), it will become less than the communication delay and hence will not yield an increased speed-up. Thus, for this problem size, the addition of further machines will not result in an increased speed up.

Table 6.7 Speed-up for Steady State Problem: 300,000 Grid Points

Number of nodes	*Node*	*Grain size*	*Task time(s)*	*Waiting time(s)*	*Speed up*
1	1	300000	378	NA	NA
2	1	160000	175	10	2.04
	2	140000	170	13	
3	1	110000	124	8	2.84
	2	100000	122	10	
	3	90000	120	15	
4	1	90000	82	10	3.78
	2	75000	80	16	
	3	75000	82	18	
	4	60000	88	10	
5	1	59000	70	10	4.72
	2	59000	65	11	
	3	59000	66	9	
	4	59000	63	12	
	5	65000	66	9	

(*Contd*)

(*Contd*)

6	1	49000	47	28	5.04
	2	49000	53	21	
	3	49000	51	21	
	4	49000	51	22	
	5	49000	54	21	
	6	55000	49	23	

Table 6.8 shows the overhead of providing the two services. The load balancing service overhead includes getting load information from all kernel entities, computing load factors and deciding the target for object migration. These overheads (especially the object migration overhead) are not small. Further, the overhead of using CORBA is also high, as Table 6.4 shows. But in spite of both these factors, P-CORBA is able to achieve significant speed ups. The main reasons for this are:

Table 6.8 Overhead of Providing Services

Service	*Overhead*
Load-balancing service	200–300 ms
Object migration service	2–3 s

- P-CORBA can dynamically migrate objects to appropriate machines and thus take care of load fluctuations in the NOW.
- P-CORBA takes into account the processing speed of the various machines, before choosing the target for migration. This is very important for proper load distribution.
- While choosing the target for migration, machines which join the pool dynamically are also considered. By addressing issues specific to parallel programming over NOWs comprehensively, the overheads (in providing services or in message sending) can be offset by the gain in performance. Thus, by smoothening load fluctuations in the NOW, P-CORBA performs well as a parallel programming platform.

P-CORBA achieves significant speed-ups for a specific class of problems over a loaded workstation cluster by overlapping computation with communication. If the grain size is too small, the communication overheads may dominate and result in sub-linear speed-ups. We conducted an experiment to determine the minimum grain size at which the overheads of providing services on a loaded workstation cluster can be tolerated and maximum speed-up obtained. Table 6.9 shows the speed-up obtained for eight machines with increasing problem size on a loaded cluster. This set of performance studies were conducted on a Linux cluster of *Acer* PCs with 3.2 GHz Intel Pentium IV dual processors and 1GB of main memory, connected by a 100 Mbps LAN.

Table 6.9 Speed-up with Increasing Grain Size: Eight Machines

S.no.	*Problem size*	*Average task time(s)*	*Average waiting time(s)*	*Single node time(s)*	*Speed-up*
1	200,000	4	6	70	7
2	300,000	7	6	104	8
3	400,000	8.5	6.5	138	9.2
4	500,000	10.5	7	191	10.91

The set up is illustrated in Fig. 6.14. In this case also, by using CPUhog, one of the eight machines was artificially loaded and the task migrated at runtime. It shows that a problem size of 200 000 grid points does not scale up and gives only sub-linear speed-ups. However, at 300000 grid points, the speed-up is exactly linear (communication time is nearly equal to computation time). However, problem sizes beyond 300000 result in super-linear speed-ups. This shows that even on a loaded workstation cluster, P-CORBA can achieve linear to super-linear speed-ups.

Another interesting experiment was conducted to determine the range of load fluctuations that can be tolerated by P-CORBA. We determined the load threshold on the target machine for task migration, a non-trivial task. In the previous experiment (eight machines + problem size 500000), a further study was conducted.

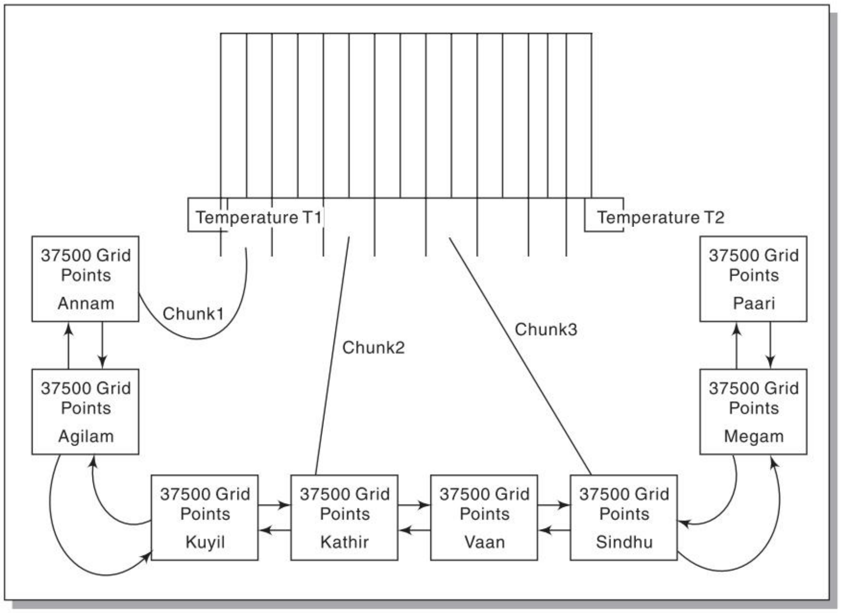


FIG. 6.14

Iterative Grid Computation (IGC) Problem

The results are tabulated in Table 6.10. The CPUhog process mentioned before was used. This program can be configured to generate many tasks and hog the CPU. In the first case, a load of 4.8 was generated by creating six tasks with the CPUhog process. The load of 6.8 was generated by using two CPUhog processes, each with four tasks. In order to generate the load of 9.5, twoPuhog processes, each with six tasks were run. The load was measured by using the top command, which measures the load as the average CPU queue length. It can be observed that up to loads of almost 8 (all these machines are dual processor machines), the node can be the target of task migration and linear or super-linear speed-up can be achieved. The threshold can be fixed at 8, as a load of 9.5 slows down the node drastically and increases the waiting time for other tasks, resulting in sub-linear speed-ups.

Table 6.10 Determining Target Load Threshold: 500000 Grid Points and Eight Machines

Serial number	*Load on target (top command)*	*Average task time(s)*	*Average waiting time(s)*	*Speed-up*
1	4.8	10.5	7	10.91
2	6.8	12	11.5	8.3
3	9.5	15	16	6.1

6.7 Suitability of CORBA: An Introspection

The important question at this stage is whether CORBA is suitable for parallel programming or not. Its advantages are two-fold. First, its support for object orientation facilitates object-based parallel computing. The authors believe that it is the right approach for parallel programming over NOWs. The object location mechanism as well as the remote activation mechanism supported by CORBA ORBs automatically, prevent the system programmer from handling low-level network details. Secondly, it handles heterogeneity in architecture and operating systems. This is an important issue for parallel programming over a NOW.

Addition of two features into CORBA could make parallel programming much easier. One is a load-balancing service, implemented as either a standalone service as suggested in the model or integrated into naming or trading services [3]. The other is a proper object migration facility, as the model suggests. Further, the drafting of AMI into the CORBA specification will ease parallel programming considerably.[19] Implementation of the Concurrency service may also be important, especially if the notion of concurrency used is similar to the one used in this model.

6.8 Conclusions

This chapter has presented P-CORBA, a model for parallel programming over CORBA that addresses issues specific to parallel computing over a NOW. The main contribution of the model is the introduction of the notion of concurrency into the CORBA domain. The model also demonstrates one way of balancing the load in a CORBA environment. The performance comparison over MPI suggests that it is a good idea to use CORBA for parallel programming over NOWs. The performance studies also show that P-CORBA can obtain linear to super-linear speed-ups for certain classes of applications. This work has established that in spite of the relatively high message sending overhead in CORBA, it can be used for parallel programming over NOWs. Further, several attempts are being made to reduce the message sending overhead in CORBA [12]. These are being incorporated into the next version of CORBA (3.0) through a real-time CORBA specification. A fast CORBA combined with the parallel programming model presented in this work could provide the best possible support for utilizing the computing power of a NOW. It would be interesting to explore the possibility of wide area parallel computing over a real-time CORBA.

Parallel programming was not the fundamental design goal of CORBA. The fact that it may be easily used in this domain shows the adaptability of the specification. This suggests possible extensibility and its applicability in various other areas too. Hence, the evaluation of CORBA for parallel programming could be an indication of the very survival of CORBA over a longer period of time.

DCOM [9] can also be used as the middleware for parallel programming. The ultimate step might be to use DCOM for Windows clusters (due to good performance) and CORBA for heterogeneous clusters. Parallel programming with DCOM and CORBA over WANs could be explored.

The integration of eXtensible Markup Language (XML) [36] with CORBA can be explored. This could facilitate the exchange of intermediate results in parallel programming through the standard for data interchange, XML. Thus, P-CORBA could provide a single parallel programming model in which code distribution is handled through CORBA and data distribution is handled through XML.

References

1. The NOW Team, Anderson, T.E., D.E. Culler, D.A. Patterson, "A Case for NOW (Network of Workstations)", *IEEE Micro*, Vol. 15, No. 1, pp. 54–64, February 1995.
2. Bal, H.E., J.G. Steiner, A.S. Tenenbaum, "Programming Languages for Distributed Computing Systems", *ACM Comput. Surveys*, Vol. 21, No. 3, pp. 261–322, September 1989.
3. Barth, T., *et al*., "Load Distribution in a CORBA Environment", International Conference on Distributed Objects and Applications (DOA), Edinburgh, Scotland, September 1999.
4. Binu, K.J., R. Karthikeyan, D. Janakiram, "DP: A Paradigm for Anonymous Remote Computation and Communication for Cluster Computing", *IEEE Trans. Parallel Distributed Systems* Vol. 12, No. 10, pp. 1–14, October 2001.
5. Brunner, R.K., L.V. Kalé, "Adapting to Load on Workstation Clusters", The Seventh Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press, Silver Spring, MD, pp. 106–112, February 1999.
6. Buttner, L., J. Nolte, W. Schroder-Preikschat, "ARTS of PEACE–A High-performance Middleware Layer for Parallel Distributed Computing", *J. Parallel Distributed Comput. 59* pp. 155–179, September 1999.

7 Carriero, N., *et al*., "Adaptive Parallelism and Piranha", *IEEE Comput.,* Vol. 28, No. 1, pp. 40–49, January 1995.

8. Coulson, G., D. Waddington, "A CORBA-compliant Real-time Multimedia Platform for Broadband Networks", International Workshop on Trends in Distributed Systems (TreDS), Lecture Notes in Computer Science, Vol. 1161, Springer, Berlin, October 1996.
9. DCOM: Distributed Component Object Model Protocol–DCOM/1.0, http://msdn.microsoft.com/library/specs/distributedcomponentobjectmodelprotocoldcom10.htm.

10. Fox, G.C., *et al.*, "Solving Problems on Concurrent Processors", *General Techniques and Regular Problems*, Vol. 1, Prentice-Hall, Engelwood Cliffs, New Jersey, 1988.

11. Gannon, D., *et al.*, "Developing Component Architecture for Distributed Scientific Problem Solving", *IEEE Comput. Sci. Eng.*, Vol. 5, No. 2, pp. 50–63, April–June 1998.

12. Gokhale, A.S., D. Schmidt, "Measuring and Optimizing CORBA Latency Over High-speed Networks", *IEEE Trans. Comput.*, Vol. 47, No. 4, pp. 319–412, April 1998.

13. Grimshaw, A.S., "Easy-to-use Object-oriented Parallel Processing with Mentat", *IEEE Comput.*, Vol. 27, No. 5, pp. 30–51, May 1993.

14. Grimshaw, A.S., W. Wulf, "The Legion Vision of a Worldwide Virtual Computer", *Comm. Assoc. Comput. Mach.*, Vol. 40, No. 1, pp. 39–45, January 1997.

15. Henning, M., "Binding, Migration and Scalability in CORBA", *Comm. Assoc. Comput. Mach.*, Vol. 41, No. 10, pp. 62–71, October 1998.

16. Henning, M., S. Vinoski, "Advanced CORBA Programming with C++", Addison-Wesley, Reading, MA, 1999.

17. Janakiram, D., T.H. Sreenivas, G. Subramaniam, "Parallel Simulated Annealing Algorithms", *J. Parallel Distributed Comput.*, Vol. 37, pp. 207–212, 1996.

18. Janakiram, D., A. Vijay Srinivas, "Object Migration in CORBA", *J. Comput. Soc. India*, Vol. 32, No. 1, pp. 18–27, March 2002.

19. Joshi, R.K., D. Janakiram, "Object-based Sub-contracting: A model for Parallel Programming on Loosely Coupled Workstations", *J. Programming Languages* 4, pp. 169–183, 1996.

20. Joshi, R.K., D. Janakiram, "Anonymous Remote Computing: A Paradigm for Parallel Programming on Interconnected Workstations", *IEEE Trans. Software Eng.*, Vol. 25, No. 1, pp. 75–90, January/February 1999.

21. Kalé, L.V., S. Krishnan, "CHARM++: A Portable Concurrent Object-oriented System based on C++", in A. Paepcke (ed.), Proceedings of OOPSLA'93, ACM Press, pp. 91–108, New York, September 1993.

22. Keahey, K., D. Gannon, "PARDIS: CORBA-based Architecture for Application Level Parallel Distributed Computation", in *SuperComputing 97*, August 1997.

23. Kunz, T., "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme", *IEEE Trans. Software Eng.*, Vol. 12, No. 4, pp. 269–284, July 1991.

24. Leppinen, M., P. Pulkkinen, A. Rautiainen, "Java and CORBA-based Network Management", *IEEE Comput.*, Vol. 30, No. 6, pp. 85–87, June 1997.

25. Litzkow, M.J., M. Livny, M.W. Mutka, "Condor–A Hunter of Idle Workstations", IEEE International Conference on Distributed Computing Systems, IEEE Press, New York, pp. 104–111, June 1998.

26. Manjula Rani, P., A. Vijay Srinivas, D. Janakiram, "Scalability Issues in CORBA", The Fifth International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000), Limerick, Ireland, June 2000.

27. MPI Forum, "MPI: A Message Passing Interface Standard", 1995.

28. Object Management Group, "CORBA Messaging Specification", OMG Document orbos/98-05-05 (ed.), 1998.

29. Object Management Group, *CORBA Services: Common Object Services Specification*, Revised edition, December 1998.

30. Object Management Group, *The Common Object Request Broker: Architecture and Specification. 2. 3. 1*, October 1999.

31. Paepcke, A., *et al.*, "Using Distributed Objects to Build the Stanford Digital Library Infobus", *IEEE Comput.*, Vol. 32, No. 2, pp. 80–87, February 1999.

32. Puder, A., K. Romer, MICO is CORBA; http://www.dpunkt.de/mico/.

33. Shivaratri, N.G., P. Krueger, M. Singhal, "Load Distributing for Locally Distributed Systems", *Computer*, Vol. 25, No. 12, pp. 33–44, December 1992.

34. Tandiary, F., *et al*., "Batrun: Utilizing Idle Workstations for Large-scale Computing", *IEEE Parallel and Distributed Technology*, pp. 41–49, 1996.

35. Wolfgang, S., *The Logical Design of Parallel Operating Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1994.

36. W3C Recommendation, "eXtensible Markup Language (XML)", http://www.w3c.org, February 1998.

37. M.V., Steen, P. Homburg and A.S. Tanenbaum, "Globe: A Wide-Area Distributed System", *IEEE Concurrency*, Vol. 7, No. 1, pp. 70-78, January-March 1999.

38. Janakiram, D., A. Vijay Srinivas and P. Manjula Rani, 'A Model for Parallel Programming Over CORBA', Tech Report, IITM-CSE-DOS-99-06, Distributed and Object Systems Lab, Dept. of CS&E, IIT Madras, Chennai, December, 1999, communicated to *Cluster Computing Journal*, Baltzer Science Publications, Netherlands, available at *http://lotus.iitm.ac.intechrep.*

39. Pasala, A. and D. Janakiram, "FlexiFrag: A Design Pattern for Flexible File Sharing in Distributed Collaborative Applications", *Journal of Systems Architecture: The Euro Micro Journal*, Elsiever Science, Vol. 44, pp. 937–954, 1998.

40 R.K., Joshi, N. Vivekananda and D. Janakiram, "Message Filters for Object-oriented Systems", *Software-Practice and Experience*, Vol. 27, No. 6, pp. 677–699, June 1997.

41. Sriram Reddy, G. and R.K. Joshi, "Filter Objects for Distributed Object Systems", to appear in *Journal of Object-oriented Programming.*

42. Puder, Arno and Kay Romer, "MICO is CORBA", *http://www.dpunkt.de/mico.*

43. Fuggetta, A., G.P. Picco and G. Vigna, "Understanding Code Mobility", *IEEE Transactions on Software Engineering*, Vol. 24, No. 5, pp. 342–361, May 1998.

44. Jul, E., H. Levy, M. Hutchinson and A. Black, "Fine-grained Mobility in the Emerald System", *ACM Transactions on Computer Systems*, Vol. 6, No. 1, pp. 109–133, February 1988.
45. Shillner, R.A., and E.W. Felton, "Mobile Network Objects", Technical Report, TR-534-96, University of Princeton, October 1996.
46. Caughey, S.J. and S.K. Shrivatsava, "Architectural Support for Mobile Objects in Large-scale Distributed Systems", International Workshop on Object Oriented Systems, 1995.
47. Narasimhan, P., L.E. Moser and P.M. Melliar-Smith, "The Interception Approach to Reliable Distributed CORBA Objects", Proceedings of the 3rd Conf. on Object-oriented Technologies and Systems, June 1997, available at *http://www.usenix.org/publications/library/proceedings/coots97/narasimhan.pdf*.

End Notes

1. This is the Object Request Broker (ORB) in CORBA.
2. This decision should not be left to the programmer as he may not be aware of the load conditions on the nodes of the NOW.
3. 'Best available machines' refers to the machines with the lowest load factor. It is a simple mathematical function of the load and processing power on a node. The lower the load factor, the higher is the load that the machine can take.
4. This task can actually be automated. In fact, the translator does precisely this function as Section 6.4.1 explains.
5. Popular ORBs such as visibroker, orbix, omniorb, and mico have not implemented this. Only IBM's Component Broker and Washington University's TAO provide implementations.
6. These guidelines are general in nature. Any service must adhere to these principles [29].

7. Server object is the client of the filter object
8. The filterable-impl class provides implementations for the attach() and detach() methods
9. The functions of this entity are discussed comprehensively in the next section.
10. This provides implementations of all the three methods of the interface.
11. This is equivalent to declaring the exception in the object's IDL due to the inheritance hierarchy.
12. This can be generated as a Universally Unique IDentifier (UUID) as in DCOM [12].
13. Since it created the new object, it has a reference to that object.
13. A number of machines are configured to use a single migrating entity.
15. Since the POA interface is standarized, adding a new method for updation is not easy.
16. If filters are used for a different purpose, then the code may have to be changed. In this case, the filters only redirect client requests. The only thing that changes for each filter is the location and possibly the return value types of the methods.
17. This is available in the source code of the object.
18. A process which deliberately hogs the CPU and generates configurable loads. Check *http://www.xmailserver.org/linux-patches/lnxsched.html* for more details.
19. The recently released CORBA specification incorporates AMI in the form of the messaging specification.

Chapter 7

Sneha–Samuham: Grid Computing Model*

7.1 Introduction

The Internet has become an attractive means of sharing information across the globe. The idea of 'grid computing' has emerged from the fact that the Internet can also be used for several other purposes such as sharing the computing power, storage space, scientific devices and software programs. The term 'grid' is chosen as it is analogous to the electric power grid where it provides consistent, pervasive, and ubiquitous power irrespective of its source [1]. The focus of this chapter is on computational grids. A computational grid is a sub-class of a general grid, wherein it contains only computing resources. The aim of management software running on a computational grid is to efficiently execute applications submitted by the user apart from providing support for heterogeneity, fault tolerance, security, etc.

There exist a considerable number of parallel computing models that can exploit coarse grain parallelism over a NOW. The significant efforts include PVM[2], MPI[3], ARC[4], DP[5], Linda[6], Piranha[7], Condor[17], etc. However, these models lack the infrastructure to extend over the Internet. There are some models which can provide global computing, such as Javelin[8], ATLAS[9], POPCORN[10], Globus[11], Legion[12], etc. Some of these

*D. Janakiram, N.V. Palankeswara Rao, A. Vijay Srinivas M.A. Maluk Mohamed models in general, and Globus in particular, address a lot of grid computing issues such as security, authentication, fault tolerance, etc. But none of the above models provides support for an adaptive parallel execution of an application by transparently splitting a task into sub-tasks of appropriate granularity.

The Sneha-Samuham[1] model provides adaptive parallel execution of tasks over a computational grid. Its strength lies in transparent splitting of a parallel task into sub-tasks of appropriate granularity, depending on the computation capability of participating nodes. Moreover, its user interface for sharing the computing resources across the Internet, makes the model user-friendly. Users can harness the computing power of 'friendly' computers over the Internet for parallel processing.

A good parallel computing mechanism can boost the performance of a computational grid to execute applications. None of the existing grid computing models supports automatic task splitting, even though it is possible for many classes of applications. These systems expect the user to submit a batch of jobs to the system and the scheduler schedules these jobs on the participating nodes. In the proposed model, automatic task splitting support has been provided for certain classes of applications. This relieves the user from the burden of task splitting related issues. Also, the user does not need to know the computation capabilities of individual nodes participating in the grid. Currently, Sneha-Samuham supports only purely data parallel scientific applications that are coarse-grained. We are extending the applicability of the model to a wider range, especially to support applications involving communicating tasks.

The rest of the chapter is organized as follows. Section 7.2 gives an overview of the 'Sneha–Samuham' model. Section 7.3 provides the design and implementation details of the model. Section 7.4 discusses the performance evaluation of the model. Section 7.5 describes some of the existing grid computing models. Section 7.6 concludes the work. It also gives new ideas for future research.

7.2 Sneha-Samuham: A Parallel Computing Model Over Grids

In the Sneha–Samuham model, a computer, over the Internet, can donate its computing power to other computers as well as use the computing power of other computers. The computer which donates its computing power is called a 'donor' and the computer which makes use of another computer's computing power is called an 'acceptor'. A node[2] can act either as a 'donor' or as an 'acceptor' or both as an 'acceptor' and a 'donor'. Ideally, the number of donors should be much higher as compared to the number of acceptors. An acceptor can use the computing power of its donors for executing parallel applications. The topology, architecture, various components and the computation model of Sneha–Samuham are explained in the following sub-sections.

7.2.1 System Topology

The Internet can be treated as an interconnection of individual LANs as shown in the Fig. 7.1. The collection of nodes within an individual LAN is referred to as a 'cluster'. A designated node in each cluster acts as a mediator between the nodes inside its cluster and the outside world.

The mediator is called a Cluster Coordinator (CC). If a cluster contains a single node, then that node itself acts as the CC for that cluster. Any node inside a cluster (including the CC) can act either as a 'donor' or as an 'acceptor' or both as an 'acceptor' and as a 'donor' or may not participate in the grid computing at all.

There are several advantages with this kind of topology. Since there is no global coordinator, the model is scalable and there won't be any single point failures. The next advantage is that every node of a cluster participating in the grid computation need not be accessible from outside, over the Internet. Since every cluster that participates in the computation, contains a CC, it is sufficient if that CC is accessible over the Internet. The machines inside a remote cluster can be accessed through the CC of that cluster. Private addressing and source Network Address Translation

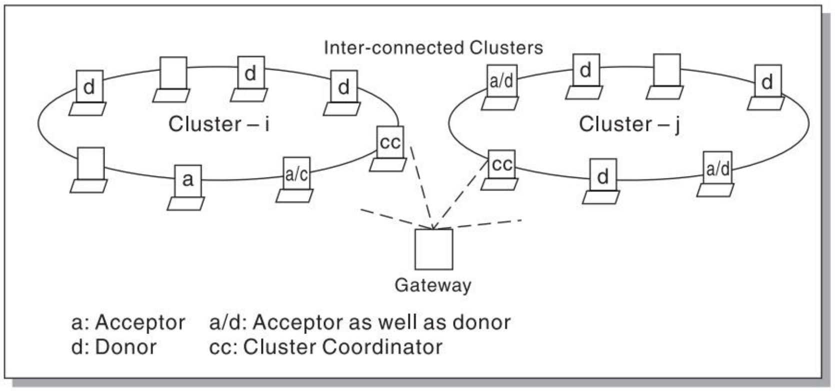


FIG. 7.1

System Topology

(NAT) of a LAN need not be disturbed while the machines of the LAN are used for grid computing. Its other advantage is that providing authentication, security and fault tolerance for grid computing becomes easy. For example, there is no need to secure a machine from other machines inside its cluster. It is enough to have a good security system in the CC to secure the machines of a cluster from outside machines.

7.2.2 Layered Architecture of Sneha–Samuham

The Sneha–Samuham model is viewed as a five-tiered architecture, with the middle three layers constituting the core. The functionality of each layer is explained in this section.

- **Computing Resources**

This layer contains various computing resources, with varying processor speeds, memory, architecture and operating system running on them. In this model, a computing resource implies, the combination of both the hardware resource and the local management software (operating system) running on it.

- **Runtime Environment**

Runtime environment interacts with local operating systems existing in the computing resources layer. It is responsible for collecting the sub-tasks from the user interface, migration and execution of these sub-tasks on the remote machines over the Internet, and getting the results back to the node, where the computation was initiated. The runtime environment of the system contains three components that run as daemons on the respective nodes: acceptor, donor and ccdaemon. A node which acts as both a donor as well as an acceptor runs both the daemons. Every CC runs a daemon called ccdaemon to coordinate between the acceptors and the donors. The node to which a parallel task is submitted should have the acceptor daemon running. Also, the nodes where the donor daemon is running can participate in the parallel computation by sharing work from acceptors. The interactions among these three daemons is governed by a well-defined protocol. The core grid computing services provided by the runtime environment can be accessed by using the APIs provided in the layer above it.

Whenever a cluster wants to participate in the parallel computation, it initially starts the ccdaemon on its CC. After a donor or an acceptor starts at a node, it registers with its local CC through the ccdaemon running on that CC.

- **Application Programming Interfaces (APIs)**

This layer contains various APIs to access the services provided by the runtime environment for grid computing. This layer is introduced to make the runtime environment application-independent. These APIs provide a standard way for the above layer (user interface), to request the runtime environment for various kinds of grid computing services, such as collecting friend CCs information, migration of a sub-task to a remote node, getting back the results of a sub-task, etc. This layer, including the runtime environment layer, provides core grid computing services and the interface to access them. Any new service provided in the runtime environment can be accessed by including the corresponding APIs in this layer. This makes the user interface transparent from the updates made at runtime environment.

- **User Interface**

This layer provides user level tools for collecting the computing resources and submitting tasks to the grid. It contains two sub-components. One is the Friend Machines Interface (FMI) to collect the resources and the other is a set of user level commands to submit applications to the grid. The FMI is an instant messenger kind of Graphical User Interface (GUI) tool, which interacts with the local cluster coordinator using the appropriate APIs in the lower layer. It provides various services to the user to aggregate the resources over the Internet for grid computing as explained below:

- The owner of a cluster can request owners of other clusters over the Internet to become a friend cluster to his/her cluster. If he/she accepts, then both the clusters become friends to each other. Each CC includes the name of the other CC in its Friend Clusters Table (FCT).
- One can accept or reject the request made by others over the Internet to make his cluster a friend to them. Protocol similar to that of 'yahoo messenger' for discovering chat friends is used.
- It displays all friend CCs of a CC and highlights the live friend CCs. Along with the friend CCs, other information such as number of donors available in each friend cluster, their GCCFs and communication latencies can also be displayed on the basis of the user's preference.
- The user or the application can select any number of donors from friend clusters for parallel computing.

If two clusters are friends to each other, any node can use the computing power of any other node in the two clusters. A cluster can have any number of friend clusters. A CC maintains a list of all its friend CCs. Whenever a parallel application is started on a node, it can make use of the machines (which are running donor daemon) that belong to all of its friend clusters for parallel processing.

In Sneha–Samuham, it is assumed that each cluster is owned by a user or a group of users. It is a realistic assumption as each

LAN is owned by a research group or an organization. They decide whether their cluster will participate in the grid computing or not. They can designate any node as a CC, an acceptor or a donor by running the corresponding daemon on it. The owner(s) can make his cluster a friend to any other cluster by using the FMI described above. A cluster including all of its friend clusters forms a 'computational grid'. A number of such grids can exist over the Internet.

Apart from FMI, the user interface also contains a set of application level commands to submit parallel applications to the grid. The set contains one command for each class of parallel applications. The functionality of these commands includes the splitting logic for that particular class of applications and makes use of the lower level APIs for accessing generic grid computing services. As the splitting logic varies for different classes of applications, each class should contain one command in this layer. A new class of applications can be supported by adding the corresponding command to the existing set of commands. The user can see the supported commands and, their description and can use the appropriate command to submit his application to the grid. However, implementing these kinds of commands is not possible for all applications as some class of applications are inherently not parallelizable.

- **Grid Computing Applications**

This layer contains the applications, which can benefit from grid computing. A grid computing application is CPU-intensive and there should be algorithms to partition the application into independently running parts. The parts must be executable remotely without much overhead, and the remote machine must meet any special hardware, software and/or other resource requirements imposed by the application. The application is more scalable, if these sub-tasks (or jobs) do not need to communicate with each other.

7.2.3 Computation Model

The computation model follows the Master-Worker [13] process model. Upon receiving the task, the user level command executes as a master process and contacts the acceptor running on that node. Then the acceptor contacts its local ccdaemon and gets the addresses and GCCFs of requested number (or available number, if requested number of machines are not available) of friend machines. The master process splits the task into sub-tasks of appropriate granularity depending upon the number of available friend machines and their current computing capabilities. Once the splitting gets over, it migrates each sub-task to the corresponding node. The donor daemon that receives the sub-task to be computed, spawns a worker process for that sub-task and returns the results after the execution is completed. The division of tasks among the available nodes is done as described in the following sub-section.

7.2.3.1 Task Splitting

The computational capability of a node over the grid can be measured from the factor called GCCF,

$$f = \frac{sm}{lt}$$

where s, m and l are processor speed, memory and average load on a node, respectively and t is the communication latency from a node, from where it received the job to be executed. The division of task among the available lightly loaded nodes is done as follows:

If there are n nodes with GCCFs, $f_1, f_2, f_3, \dots f_n$.

Let, $F = f_1 + f_2 + f_3 + \dots f_n$

And if the total task size is G, then

Grain size assigned to the first node, $g_1 = \left(\frac{f_1}{F}\right) G$

Grain size assigned to the second node, $g_2 = \left(\frac{f_2}{F}\right) G$

Grain size assigned to the n^{th} node, $g_n = \left(\frac{f_n}{F}\right) G$

The above task distribution is applicable only to data parallel applications. Normally in data parallel applications, a functionality will be executed on a large chunk of data. This data can be divided into several parts and can execute the same functionality on individual chunks. Each chunk could be of different size. This technique may have to be extended to apply for task parallel applications.

7.3 Design and Implementation of the Model

The middle three layers, viz. the runtime environment, APIs and user interface, form the core of the Sneha-Samuham model. The following sub-sections explain the designs of these three layers and implementation details of the model. In the following discussion, local machine with respect to some machine, means the one which resides in the same cluster. For example local CC means the CC of that cluster.

7.3.1 Runtime Environment

The collection of acceptors, ccdaemons, donors and their interactions form the runtime environment of the system. The interactions among these three daemons as well as their communication with the upper layer are governed by a well-defined protocol. The following sub-sections describe the design of each of these daemons and their interactions with other components of the system.

7.3.1.1 Cluster Coordinator (CC)

The CC in the Sneha-Samuham model acts as a resource collector. A cluster can be a friend to other clusters over the Internet. The CC of each cluster maintains a table called FCT. Each entry in the FCT contains the address of a friend CC. The FCT of every cluster is updated whenever a new CC is added to the list or an existing CC relinquishes the friendship with that cluster. The interaction between a ccdaemon and FMI will do this. It also maintains a pending requests list, whose entries are the requests that came from other CCs for friendship with this cluster and which are not responded by the user (or owner) of this cluster.

Apart from the friend clusters' information, a CC also maintains lists of its local donors and of its local acceptors. Whenever an acceptor from a friend cluster asks for donors, it returns the addresses of the nodes stored in the donors' list along with their current computational capabilities. The GCCF defined in the previous section represents a node's current computational capability.

A CC interacts with its FMI, local acceptors, local donors, and with other CCs. In the ***INIT*** state, ccdaemon initializes its data structures. In the ***LISTEN*** state, it waits for messages from the other daemons with which it is allowed to interact. If it receives any message from the FMI, friend CC, local acceptor or local donor, then it changes its state to ***FMI Msg RECVD***, ***Other CC Msg RECVD***, ***Acceptor Msg RECVD*** or ***Donor Msg RECVD***, respectively and services the message. Any error message in these states causes the state to be changed to ***ERROR*** and the appropriate action will be taken. Various kinds of messages could come from each of the above-mentioned daemons. For example, from FMI, the message could be to request a specific CC to become a friend to it or a response to the request from another CC, forwarded by it. For each message, the ccdaemon executes a different service routine to service the request. The description of various messages exchanged among these daemons and the actions taken on reception of these messages are out of the scope of this chapter.

7.3.1.2 Acceptor

The acceptor receives tasks submitted by the user interface and executes them on its friend machines. It gets the global computing resources from its local ccdaemon. Since users can submit more than one parallel task to an acceptor, it maintains a table of all submitted tasks. This table is called a Task Table (TT). Each entry in the TT contains a task-ID and the corresponding sub-task IDs after the task is split into sub-tasks. Each entry is updated with the target CC and donor addresses after the sub-tasks are migrated.

The acceptor interacts with its local CC, local donors (in case it submitted jobs to them) and with the user interface. In the ***INIT*** state, the acceptor daemon initializes its data structures and registers with its local CC. In the ***LISTEN*** state, it waits for messages from the other processes. If it receives a message from any of the user processes, its local CC, or its local donor, then it changes its state to ***UP Msg RECVD, CC Msg RECVD*** or ***Donor Msg RECVD***, respectively and services the message. If any error occurs in this process, the state is changed to ***ERROR*** and the appropriate action is taken.

7.3.1.3 Donor

A donor daemon registers with its local ccdaemon and gives replies to the requests made by its ccdaemon such as current GCCF of the machine, etc. If the current GCCF is above some threshold, it receives sub-tasks from the acceptors of its friend machines, executes them locally, and sends results back to the acceptor if it is local or through its CC if it is from an outside cluster. In order to store information of all the sub-tasks that are running on this node, it maintains a table called Sub-Task Table (STT). Each entry in the STT contains a sub-taskID, taskID generated by its parent node,[3] source CC and acceptor addresses from where the sub-task has been migrated. This information is required to send the results back after the execution of a sub-task is finished. If the sub-task is from a local acceptor, then its source CC address is zero and the result of the sub-task can be directly sent to the acceptor.

A donor daemon interacts with its local CC, local acceptors and the Worker Processes (WPs) spawned by it for executing sub-tasks. In the ***INIT*** state, it initializes its data structures and registers with its local ccdaemon.

In the ***LISTEN*** state, it waits for the messages from the other processes with which it could interact. If the donor daemon receives a message from its ccdaemon, local acceptor or from its worker process, then it changes its state to ***CC Msg RECVD***, ***Acceptor Msg RECVD*** or ***WP Msg RECVD***, respectively and services the message. If any error occurs in this process, the state is changed to ***ERROR*** and the appropriate action is taken.

7.4 Performance Studies

The neutron shielding simulation [14] application has been chosen for studying the performance of the Sneha–Samuham grid computing model. It is a nuclear physics application, in which a beam of neutrons is delivered in the experiment. When this neutron beam strikes a lead sheet of certain thickness perpendicularly, it is important to predict the number of neutrons that can penetrate from the other side of the lead sheet. This application predicts that number by using the Monte–Carlo simulation. More details about this application can be found at [14]. For all simulation experiments, a lead sheet of 5 units thickness is assumed. The number of neutrons will be a power of 10 in each experiment. In practical situations, the number of neutrons depends upon the type of experiment and the amount of time for which that experiment has to be conducted. The amount of computing power required is directly proportional to the number of neutrons. In the following discussion, homogeneous and heterogeneous machines (or clusters) are categorized with respect to processor speed only, that is, in a cluster of homogeneous machines, all the machines will have the same processor speed.

7.4.1 Power of Sneha–Samuham: Institute-wide Grid

Table 7.1 shows the power of Sneha–Samuham in executing coarse-grain parallel applications such as the neutron shielding simulation. The first column of the table shows the number of neutrons (problem size) for each experiment. The time taken for this application by a single machine and the time taken on the grid of three clusters using Sneha–Samuham has been measured. The values in the second column are the execution times for the corresponding problem size given in the first column, using a single machine of processor speed 367.5 MHz. The values in the third column are achieved by a grid of 14 machines, distributed across three clusters with varying processor speeds ranging from 267 MHz to 2400 MHz. The fourth column shows the speed-up achieved by the grid when compared to the single machine.

Table 7.1 Institute-wide Grid Performance

Prob. size	*Time taken (sec)*		*Speed-up*
	Single machine(s)	*Grid (g)*	*(s/g)*
10^{10}	49590.55 (= 13.77 hrs)	148188 (= 0.41 hrs)	33.46
10^9	5056.07	152.14	33.23
10^8	526.29	20.47	25.71
10^7	49.32	6.22	7.93
10^6	4.92	5.24	0.94

The results show the efficiency of the grid computing model for highly computation-intensive applications. It is clear from the first row of the table that an application which takes around 14 hours on a single machine can be finished in around 25 minutes on a small grid of 14 machines. So the applications which take days and years to execute on a single machine can very efficiently be executed in a fraction of an hour or in few hours using sufficiently large grids. Here the Sneha–Samuham model has an advantage in that it divides the task according to the computation and communication capabilities of the participating nodes.

However, for less computation-intensive applications, the grid takes more time than a single machine. This is due to the fixed overhead of the grid for collecting GCCFs of participating nodes, splitting the task and sending the sub-tasks to the respective nodes. The last row (problem size of 10^6) shows this result, where the fixed overhead of Sneha–Samuham dominates the execution time of the application. Hence, to benefit from Sneha–Samuham, the application must be sufficiently computation-intensive to compensate the fixed overhead caused by the grid. Normally, the execution times of grid computing applications are several orders of magnitude greater than the values shown in the table.

7.4.2 Fixed Overhead caused by Sneha–Samuham

Table 7.2 shows the fixed overhead caused by Sneha–Samuham when compared to the MPI equivalent of the same application.

This experiment has been conducted on a single cluster of five homogeneous machines. The first column shows the number of neutrons (problem size) in that experiment. The values shown in the second and third columns are execution times in seconds achieved using Sneha–Samuham and MPI, respectively for the corresponding problem sizes. The last column is the difference of these two execution times, which is the overhead caused by Sneha–Samuham. The major overhead is due to the collection of GCCFs before starting the application, but the same does not exist in MPI. However, this overhead is negligible as it is much lesser around 1 second) than the computation times of parallel applications.

Table 7.2 Overhead of Sneha–Samuham

	Time taken (sec)		
Prob. size	*Sneha–Samuham(s)*	*MPI (m)*	*Difference (s – m)*
10^{10}	10432.11	10430.93	1.18
10^{9}	1042.33	1041.18	1.15
10^{8}	105.85	104.70	1.15
10^{7}	11.85	10.59	1.26
10^{6}	1.96	1.24	0.72

7.4.3 Advantages of Sneha–Samuham over MPI

Table 7.3 illustrates the advantage we get to compensate for the cost of fixed overhead. In this case, the experiment has been conducted on a single cluster of five nodes with different processor speeds ranging from 367.5 MHz to 2400 MHz. The second column shows the single node equivalent running time for that application. This value is equal to the ratio of summation of the times taken by individual nodes for that application and the number of nodes. This is required for measuring the speed-up on a heterogeneous cluster. The second and third columns show the execution times in seconds achieved using Sneha–Samuham and MPI, respectively for the corresponding problem sizes. Please note that these values are not the same as that of the previous table (Table 7.2) as the machines in this cluster are heterogeneous. The fifth and sixth columns show the speed-ups achieved by Sneha–Samuham and MPI, respectively.

Table 7.3 Advantage of Sneha–Samuham over MPI on Heterogeneous Cluster

Prob. size	*Time taken (sec)*			*Speed-up of Sneha–Samuham $ss = a/s$*	*Speed-up of MPI $sm = a/m$*	*Difference in speed-ups $(ss - sm)$*
	Single node equivalent (a)	*Sneha–Samuhan (s)*	*MPI (m)*			
10^{10}	20355.36	3072.16	10611.49	6.63	1.92	4.71
10^{9}	2085.99	329.52	1067.60	6.33	1.95	4.38
10^{8}	206.82	33.48	107.21	6.18	1.93	4.25
10^{7}	20.43	4.13	10.74	4.95	1.90	3.05
10^{6}	2.04	1.35	1.08	1.51	1.89	–0.38

Since the Sneha–Samuham model splits the task according to the GCCF of each machine, it achieves better speed-up than its MPI counterpart, where the task is distributed in equal granularities among the machines. However, for very low computation times, Sneha–Samuham cannot perform well due to the fixed overhead. The last row of Table 7.3 shows such a case.

7.4.4 Results on a Wide Area Grid

The above experiments were conducted on an institute-wide grid. We have used some nodes from the Indian Institute of Information Technology, Bangalore, to form a wide area grid. For this particular experiment, only two machines from the remote cluster (Bangalore) were used, while eighteen machines from IIT, Madras were used. The basic idea is to demonstrate the feasibility of using geographically dispersed nodes for grid computing. Hence, only two remote nodes were chosen. We are planning to conduct larger experiments by taking an equal number of remote nodes, after addressing other issues such as security.

The fourth column of Table 7.3 gives the execution times of the corresponding application sizes on a single most powerful node out of all nodes which participated in the grid computation. The fifth and sixth columns show the speed-ups of the wide-area grid and the Institute grid with respect to the single powerful machine. For sufficiently large problem sizes, the wide area grid (of 20 machines) is around 15 times more efficient than a single powerful machine.

Table 7.4 Results on a Wide Area Grid

	Time taken (sec)			*Speed-up*	
Prob. size	*Wide area grid (g)*	*Institute grid (l)*	*Single powerful machine(s)*	*Wide area grid (s/g)*	*Institute grid (s/l)*
10^{11}	5800.87	7361.42	87016.75	15.00	11.82
10^{10}	598.11	643.31	8700.17	14.55	13.52
10^{9}	58.96	64.66	867.94	14.72	13.42
10^{8}	6.24	6.79	86.74	13.90	12.77
10^{7}	1.50	1.31	8.73	5.83	6.72

It shows that, even with just two remote machines, it is possible to achieve improved performance. As the communication delay increases with the use of geographically dispersed nodes as compared to the use of nodes from the same cluster, the grain sizes required to achieve significant speed-ups will also increase. It would also become difficult to achieve load balancing than over a single cluster, as the overheads of collecting load information will increase. These effects may be pronounced for applications which involve inter-task communication, as the computation to communication ratio assumes significance.

7.5 Related Work

Javelin [8], ATLAS [9] and Popcorn [10] are Java-based batch processing grid computing models. These models follow the broker kind of architecture wherein the user submits a batch of jobs to the intermediate nodes and these nodes, in turn, schedule the jobs at the available nodes participating in the grid. These models expect the user to split the task into sub-tasks and submit them to the system. Also, since these models are Java-based, performance may become a cause for concern.

Globus [11] provides low-level services such as resource location, resource management, communication and security, on which higher level metacomputing software can be built. Instead of competing, our model complements the Globus features. As Globus provides standard protocols for resource location, resource management, communication and security at lower layers, the proposed parallel computing model can be built on top of these layers. Like Globus, Legion [12] also provides grid computing middleware services, but it is built on the basis of object-oriented principles.

Condor-G [15] and Nimrod/G [16] are application level schedulers for parameter sweep applications. These two are extensions of the famous cluster computing models Condor [17] and Nimrod [18], respectively, and built on top of the Globus middleware services. Condor-G tries to schedule the tasks near the data required by a task. Nimrod/G tries to meet the soft real-time deadlines and at the same time, it tries to keep the cost of computation as low as possible.

7.6 Conclusions

A scalable architecture for parallel computing over grids has been discussed. This is the first attempt made to provide an "instant messenger" kind of friend machines interface to form computational grids over the Internet. The present user interface is minimal and supports only a single class of parallel applications. A number of scientific applications fall into this category of purely data parallel and coarse grained applications: parallel SA [20], parallel image rendering, distributed iterative solvers, etc. We are extending the applicability of the model to a wider range of applications, especially to support applications involving communicating tasks.

We have demonstrated the feasibility of using geographically dispersed nodes and forming a grid for parallel computation.

Further experiments are underway to use a greater number of remote nodes and to determine the grain size at which linear speed-ups can be achieved.

The present Sneha–Samuham model lays more emphasis on the parallel processing of grid applications. It has not addressed the other issues in grid computing such as fault tolerance, heterogeneity, security, etc. Many of these functionalities can be provided by using the toolkits provided by Globus. Work is under progress to provide these features. Our future directions include generalizing the model by introducing suitable language level constructs such as the ones discussed in [4] and [19] for expressing coarse-grained parallel applications.

References

1. Janakiram, D., A. Vijay Srinivas and P. Manjula Rani, “A Model for Parallel Programming Over CORBA”, *Journal of Parallel and Distributed Computing*, Vol. 64, No. 11, pp. 1256–1269, November 2004.
2. Sunderam, V.S., “PVM: A Framework for Parallel Distributed Computing”, *Concurrency: Practice and Experience*, Vol. 2, No. 4, pp. 315–339, December 1990.
3. M.P.I. Forum, “MPI: A Message Passing Interface Standard”, *http://www.mpi-forum.org*.
4. Joshi, R.K. and D. Janakiram, “Anonymous Remote Computing: A Paradigm for Parallel Programming on Interconnected Workstations”, *IEEE Transactions on Software Engineering*, Vol. 25, No. 1, pp. 75-90, January-February 1999.
5. Binu, K.J., R. Karthikeyan and D. Janakiram, “DP: A Paradigm for Anonymous Remote Computation and Communication for Cluster Computing”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12, No. 10, October 2001.

6. Carriero, N. and D. Gelernter, "The S/Net's Linda Kernel", *ACM Transactions on Computer Systems*, Vol. 4, No. 2, pp. 110–129, May 1986.

7. Carriero N., E. Freeman, D. Gelernter and D. Kaminsky, "Adaptive Parallelism and Piranha", *IEEE Computer*, Vol. 28, No. 1, pp. 40–49, January 1995.

8. Neary, M.O., B.O. Christiansen, P. Capello and K.E. Schauser, "Javelin: Parallel Computing on the Internet", *Future Generation Computer Systems*, Vol. 15, pp. 659–674, October 1999.

9. Baldeschwieler, J., R. Blumofe and E. Brewer, "ATLAS: An Infrastructure for Global Computing", Seventh ACM SIGOPS EuropeanWorkshop on System Support for Worldwide Applications, Connemara, Ireland, pp. 165–172, September 1996.

10. Camiel, N., S. London, N. Nisan and O. Regev, "The POPCORN Project: Distributed Computation over the Internet in Java", Sixth International World Wide Web Conference, April 1997.

11. Foster, I. and C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit", *International Journal of Super Computer Applications and High Performance Computing*, Vol. 11, No. 2, pp. 115–128, Summer 1997.

12. Grimshaw, A. and W. Wulf, "The Legion Vision of a World-wide Virtual Computer", *Communications of the ACM*, Vol. 40, No. 1, pp. 39–45, January 1997.

13. Andrews, Gregory R., "Paradigms for Process Interaction in Distributed Programs", *ACM Computing Surveys*, Vol. 23, No. 1, pp. 49–90, March 1991.

14. Cheney, Ward and David Kincaid, '*Numerical Mathematics and Computing*', Fourth edition, 1999.

15. Frey, J., T. Tannenbaum, I. Foster, M. Livny and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-institutional Grids", Proceedings of the Tenth International

Symposium on High Performance Distributed Computing, pp. 55–66, 2001.

16. Buyya, Rajkumar, David Abramson and Jonathan Giddy, "Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid", Proceedings of the HPC ASIA '2000, the Fourth International Conference on High Performance Computing in Asia-Pacific Region, Beijing, China, IEEE Computer Society Press, USA, 2000.
17. Litzkow, Michael J., Miron Livny and Matt W. Mutka, "Condor–A Hunter of Idle Workstations", Proceedings of Eigth International Conference on Distributed Computing Systems, San Jose, California, June 13–17, pp. 104–111, 1988.
18. Abramson, D., Sosic R., Giddy J. and Hall B., "Nimrod: A Tool for Performing Parametrised Simulations using Distributed Workstations", The Fourth IEEE Symposium on High Performance Distributed Computing, Virginia, August 1995.
19. Binu, K. J. and D. Janakiram, "Integrating Task Parallelism in Data Parallel Languages for Parallel Programming on NOWs", *Concurrency: Practice and Experience*, Vol. 12, No. 13, pp. 1291–1315, November 2000.
20. Janakiram, D., T.H. Sreenivas, and G. Subramaniam, "Parallel Simulated Annealing Algorithms", *Journal of Parallel and Distributed Computing*, Vol. 37, No. 2, pp. 207–212, September 1996.

End Notes

1. In Sanskrit, Sneha–Samuham means 'a group of friends'.
2. The terms 'workstation', 'computer' and 'node' have been used interchangeably in this chapter.
3. Parent node of a task is the node, where the task is actually created.

Chapter 8

Introducing Mobility into Anonymous Remote Computing and Communication Model*

8.1 Introduction

Clusters are formed by exploiting the existing computing resources on the network to work together as a single system. Thus they eliminate the need for supercomputers by providing better price-performance ratio and fault tolerance as compared to the traditional mainframes or supercomputers. The availability of high-speed networks and high-performance workstations has made NOWs ideal for parallel computing. A few proposals like the Condor [1], NOW [2] and Batrun [3] had been proposed earlier for improving the utility of workstation clusters which are connected by using a wired network. However, these clusters are formed only by using homogeneous devices.

Parallel programming on workstation clusters mostly follows the COP model in which processes communicate via message passing or shared memory abstractions. This model is not suitable for loosely coupled open-ended workstations because of the heterogeneity in architecture and operating systems, dynamic load fluctuations on machines, variations in machine availability and the failure susceptibility of networks and workstations. In order to

*M. A. Maluk Mohamed, A. Vijay Srinivas, D. Janakiram address these issues, two models, namely ARC [4] and DP [5] were recently proposed.

The rapid advances made in technology recently have made mobile devices more powerful (in terms of computing) and ubiquitous (in terms of connectivity). However, it is becoming essential to support seamless computing and communication for mobile users, which would offer flexibility and increased information availability. In addition, it helps in exploiting the capacities of the distributed information over the global network, which could be accessed by the user at any time without regard to the location or mobility. There are many trends that point to the increasingly widespread use of mobile devices in contrast to the static nodes. For example, it is estimated that the number of wireless Internet users is likely to grow from 35.0 per cent to 55.2 per cent by 2007, which indicates that the number of mobile hosts (MHs) are going to be more than the static nodes that are connected to the network [6]. These factors imply that in addition to static nodes, mobile nodes constitute a major part of the cluster. This results in the emergence of the essential future paradigm, namely mobile cluster computing (MCC).

In addition, the idea of integrating mobile devices with the back-end parallel computing cluster will be essential for many classes of parallel applications such as weather forecasting, earthquake predictions, etc., wherein the mobile devices can act as data collectors. In such cases, the collected data might require some kind of processing before being stored at the back-end databases. With adequate software support, these classes of applications can handle geographically independent high-performance computing requests from mobile clients. There are many similar applications like image rendering on a battlefield, surveillance and other military applications that could benefit from the proposed mobile cluster model.

Technological growth is transforming the regular wired structure in most of the university campuses to a wireless structure. This results in a situation wherein there could be a higher number of powerful mobile devices than the static ones. In such environments, it is very clear that a significant amount of idle computing power is available on these devices. Thus when these idle computing powers are harnessed along with the idle computing powers of the static nodes, it could provide an immense platform for executing scientific applications for research purposes, which is very common in universities. This provides greater need for making mobile devices to be a part of the cluster computing model.

To the best of our knowledge, only three proposals have come out on MCC, namely [7], [8] and [9]. However [7] and [9] do not discuss how parallel programming on the mobile cluster could be done. The cluster of [7] just defines and analyses the potential application environment of MCC, and gives the generic architecture of an MCC. On the other hand, [9] gives the basic architecture of MCC, which uses IPv6 as a primary protocol, which relies on network level mechanisms. Reference [8] discusses the prototype of a mobile cluster model, namely hybrid cluster computing based on their framework with JAVA mobile objects and the mobile IP. Reference [8] addresses the mobile cluster by making the mobile host the coordinator and other participating static nodes, donors of computing power. However, it neither addresses mobility of the mobile hosts nor harnesses the idle computing power of the MHs. Further, [8] also does not consider the scenario where an MH acting as a cluster coordinator moves to a different cell.

In this chapter, we present the Moset, an Anonymous Remote Mobile Cluster Computing (ARMCC) paradigm for parallel programming on a mobile cluster, that consists of both static and mobile nodes. The model addresses the research issue of harnessing the idle computing power of both the mobile and static nodes for parallel computing. It also helps in providing the resource poor device with the required computing power to execute any highly computer-intensive application at any time from anywhere. Moset is an extension of the ARC model [4] over distributed mobile systems. The proposed novel paradigm provides transparency to the mobility of nodes, distribution of computing power and heterogeneity of network architecture, thus providing better anonymity to applications. We also present a detailed case study for the proposed model by using a computation-intense application of image rendering. Our model was successfully implemented on top of a reliable multicast protocol for distributed mobile systems [10].

The rest of the chapter is organized as follows. Section 8.2 gives the key issues involved in parallel computing on mobile clusters. Section 8.3 briefly discusses the design principles and gives an overview of the proposed Moset model. Section 8.4 discusses the computational model of Moset. Section 8.5 discusses the implementation of the Moset. Section 8.6 describes the image rendering application briefly and gives the performance analysis of the model, while Section 8.7 concludes the chapter with summary and future directions.

8.2 Issues in Mobile Clusters and Parallel Computing on Mobile Clusters

There has been an increasing interest in the use of clusters of workstations connected together by high speed networks for solving large computation-intensive problems. The trend is mainly driven by the cost-effectiveness of such systems as compared to large multiprocessor systems with tightly-coupled processors and memories. However, the recent proliferation of mobile devices and advancement in wireless connectivity has made parallel computing on mobile clusters a feasible proposal. Mobile devices can be part of the cluster by playing several unique roles. They can be used as a front-end to the cluster functionality, such as submitting a job, managing processes, or viewing statistics. In case of an MH which has very poor computing power, the device must be able to utilize the cluster seamlessly to access the computational power. However, the MH can also be a contributor of computing power to the cluster, in the case of devices such as laptops which have a substantial amount of computing power equal to their static counterparts. Distributing computing power in a cluster consisting of a network of heterogeneous computing devices represents a very complex task. However, it becomes complicated when mobile devices are also a part of it.

There are several key issues that distinguish parallel computing on mobile clusters from that of the traditional workstation clusters, namely. These are:

- Asymmetry in connectivity,
- Mobility of nodes,
- Disconnectivity of mobile nodes,
- Timeliness,
- Changing loads on the participating nodes,
- Changing node availability on the network,
- Differences in computing capabilities and memory availability, and
- Heterogeneity in architecture and operating systems.

8.2.1 Asymmetry in Connectivity

The traditional cluster computing models do not face the problem of heterogeneity in the network connection as the entire set of workstations that are participating in the clusters are connected only by the wired network. Wireless networks deliver much lower bandwidth than wired networks and have higher error rates. Mobile devices are characterized by high variation in the network bandwidth, which can shift from one to four orders of magnitude, depending on whether it is a static host or a mobile host, and on the type of connection used at its current cell. Thus the programming model must be able to distinguish among the types of connectivity and provide flexibility for easy variation of the grain size of the task to account for the variations in bandwidth. However, these systems are suitable only for coarse grain level parallelism due to the communication overhead.

8.2.2 Mobility of Nodes

Due to mobility of nodes, the notion of locality becomes important as users move from one cell to another. The locality becomes important as the change in the mobile node's location means a change in the route to that node and in the consequent communication overhead. The ability to change locations while being connected to the network increases the volatility of some of the information. Static data could become mobile in the context of mobile computing. As a node moves, nearby information servers get farther away and should be replaced by closer ones offering the same or more relevant contextual information. Traditional computers do not move, as a result of which information that is reliant on location can be configured statically, such as the local DNS (Domain Name Service) server or gateway, the available printers, and the time zone. A challenge for mobile computing is to define this information intelligently and to supply the means to locate configuration data appropriate to the present location. Mobile computing devices need to access more location-related information than stationary computers if they are to serve as ubiquitous guides to a user's environment. As the mobile device moves and as the speed of motion changes, the quality of the network link and of other available resources might change significantly. Thus, the system should be able to adjust according to the changing conditions. For example, when an MH which has taken the task moves from one cell to another, then the system still needs to track these MHs.

8.2.3 Disconnectivity of Mobile Nodes

The periods of disconnectivity of nodes in static networks are usually treated as faults. However, in the context of mobile nodes, the disconnectivity may be due to roaming and hence enter an out-of-coverage area or voluntary disconnection (doze mode) to save battery power.

8.2.4 Timeliness Issue

Timeliness refers to the delay that is taken for the mobile device to regain its full state when it moves from one cell to the other or after reentering a coverage area after disconnection. Timeliness issue is an important issue especially in real-time systems. Whenever a mobile host moves from one cell to the other, it is associated with a hand-off, to ensure that data structures related to the mobile host are also moved to the new connecting point, the

Mobile Support Station (MSS). This involves an exchange of several registration messages. This may cause some delay and it should be fast enough to avoid loss of message delivery. In addition to this, there is a possibility that the mobile host could move out of coverage after accepting the task for execution. These issues need to be addressed with respect to the mobile cluster model.

8.2.5 Changing Loads on the Participating Nodes

When using workstations for executing parallel applications, the concept of ownership is frequently present. Workstation owners do not want their machines to be overloaded by the execution of parallel applications, or they may want exclusive access to their machines when they are working. Reconfiguration mechanisms are thus required to balance the load among the nodes, and to allow parallel computations to co-exist with other applications. In order to overcome these problems, some dynamic load balancing mechanisms are needed. There are differences in loads among the nodes due to the multi-user environment, and when an application is run on a heterogeneous cluster. In these cases, it is important to balance loads among the nodes to achieve sufficient performance. As static load balancing techniques would be insufficient, dynamic load balancing techniques based on runtime load information would be essential. It would be difficult for a programmer to perform load balancing explicitly for each environment/application, and automatic adaptation by the underlying runtime is indispensable. This gets aggravated when mobile devices are part of the cluster.

8.2.6 Changing Node Availability on the Network

In traditional distributed systems, nodes keep leaving and joining the system dynamically. The joining and leaving of nodes may be due to either node failure or link failure. However, the system must be smart enough to continue with the computation. The availability of node becomes more fuzzy in a distributed mobile computing scenario as the availability is also affected by the movement of the nodes. It is possible that the node may enter an area which is not under the coverage area of any MSS. It is also possible that node availability is transient with respect to the execution of the program. While a mobile node is computing a sub-task, it can go out of coverage and enter back into the coverage area before the completion of the execution of the program.

8.2.7 Difference in Computing Capability and Memory Availability

As each host may have different capabilities (such as memory) and different processing powers, it is essential to allocate tasks to the nodes on the basis of their capabilities and processing power. MHs may especially have lower computing power and memory in contrast to their static counterparts.

8.2.8 Heterogeneity in Architecture and Operating Systems

Although it is reasonable to assume that a new and stand-alone cluster system may be configured with a set of homogeneous nodes, there is a strong likelihood of upgraded clusters or networked clusters having nodes with heterogeneous operating systems and architectures. As discussed in [4], the operating system heterogeneity could be handled through distributed operating systems. However, it will be non-trivial to handle architectural heterogeneity, since the executable files are not compatible among architectures.

The issues discussed in this section make parallel programming on mobile clusters difficult. With the issue of mobility and other constraints associated with mobile devices, the management of distribution at the programming level further hardens the task. The existing cluster computing models solve only a subset of these issues. None of the earlier work in mobile clusters discusses these except for the timeliness issue which was discussed in Reference [7].

8.3 Moset Overview

The basic principle with which the Moset model was designed was to abstract out the heterogeneity of the constituting devices from the user. The user is made transparent to the hardware, bandwidth, operating system and other heterogeneity existing below the kernel. The user is given freedom to avail of any computing power required for his application, without being concerned about whether he is working with a constrained device or not. Moset is designed with clear separation between the administration functionality and the user functionality. It is the function of the administration to install and maintain the system. Once the system is deployed, the user needs only to use the APIs to interact with the system for performing parallel computing. Figure 8.1 illustrates the basic architectural overview of the proposed Moset model.

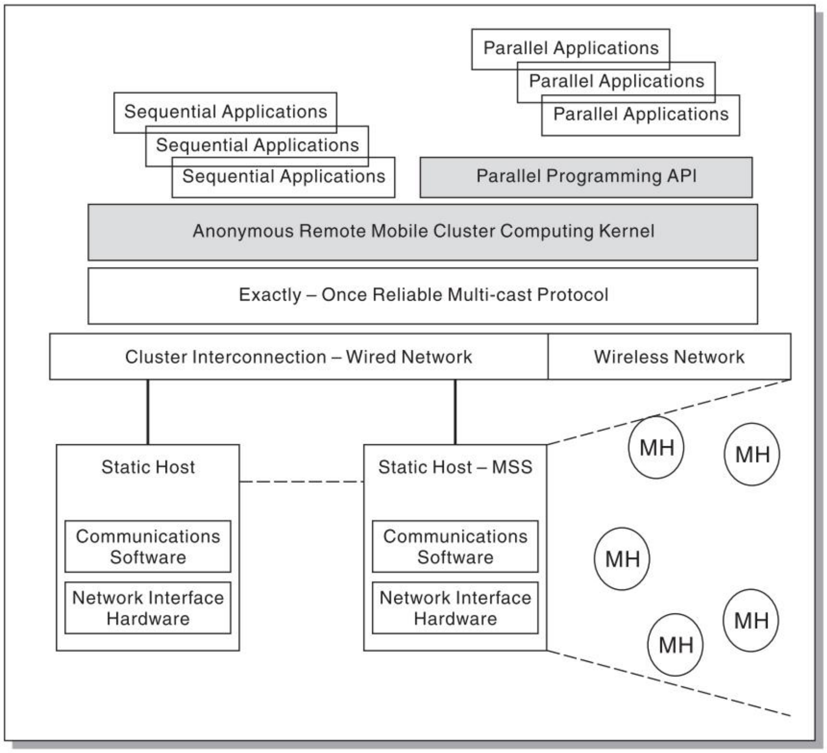


FIG. 8.1

Moset Architecture

The MSS which is a static node covers a geographical region, namely the cell. Mobile nodes which are within that cell will be under the control of that MSS, and all communications from or to the mobile nodes in the cell can be made only through the associated MSS. In our model, the MSS aggregates the computing resources which are within its cell and presents them to the distributed system as a set of its own resources. The nodes which are participating in the cluster computing are grouped on the basis of the memory capability of the nodes. The nodes which are participating in the Moset kernel spawn their computing entity to the coordinator of the system, on the basis of their capability. The entire data which needs to be processed is multi-cast to all the participating nodes in the particular group on the basis of the size of the data. The runtime system of the kernel decides on the anonymous node on which the task is to be executed on the basis of its capability. The details of the architecture are described in Section 8.5.

Unlike the nodes in a traditional distributed system, the mobile nodes cannot maintain high levels of availability or reliability due to wireless connectivity. Hence, in order to achieve reliable delivery of the data, considering the constraints of the mobile devices, Moset is built over an exactly-once reliable multi-cast protocol.

8.4 Moset Computation Model

The Moset computational model is designed in such a way that it can handle the heterogeneity, fault tolerance, dynamic load balancing and computing power availability. The dynamic load on the participating systems and the nodes and link failures make the traditional cluster computing model unsuitable for parallel programming on MCC. These issues were effectively handled in [4]. However, the model does not address the mobile device participation in computation and the issues related to it. The Moset model is aimed at integrating the mobile devices with the static nodes to form a mobile cluster, and at harnessing the idle computing power of static and mobile nodes to utilize them for parallel computing.

8.4.1 Cluster Sub-groups

In our model, we use a notion of cluster sub-groups (or sub-groups), based on the memory capability of the nodes. A cluster sub-group refers to the characteristics of the task submitted to that sub-group. Each host (static and mobile) based on its capabilities, joins the respective sub-groups. For example, sub-group LOW may refer to those tasks having memory requirements <10 MB. A MODERATE sub-group may have tasks having memory requirements <50 MB. A HIGH cluster sub-group may have tasks having memory requirements <100 MB. Tasks with memory requirements >100 MB may be in sub-group VERY HIGH.

8.4.2 Horse Power Factor and Dynamic Load Balancing

As each host may have different capabilities (such as memory) and different processing power, it is essential to allocate tasks to the static hosts and MHs on the basis of their capabilities and processing power. In order to incorporate this, each host is allocated an integer called HPF [4], which is a measure of the computing power of a machine, the load on the machine and the network bandwidth of the communication channel. Machines in the network are normalized by a benchmark program to obtain a relative index of the machine, which is a static factor. The dynamic HPF of a machine is obtained by using this static relative index, the load on the machine and the communication bandwidth with which the machine is connected to the network. This dynamic factor is normalized as a factor that represents the number of entities that it could compute. When a host has HPF h, then h computing entities are allocated to the host. For example, if hosts A and B have h_1 and h_2 as their respective HPFs, then the time taken by A to compute h_1 amount of a task is approximately equal to the time taken by B to compute h_2 amount of the same task. Abstracting the heterogeneity in this way makes parallel processing viable on unevenly loaded heterogeneous machines.

The dynamic communication bandwidth is not taken into consideration in HPF, as measuring dynamic bandwidth may lead to overhead. Also, to the best of our knowledge no technique has come out with a solution which could measure the dynamic bandwidth exactly without introducing significant overheads. This is clear from the fact that schemes like PathMon [11] (which is a relatively better scheme as compared to the other techniques like pathchirp, pathload, etc.) requires about 0.25 seconds to report the available bandwidth in a wired network, and it could be even longer in a wireless channel. This also does not guarantee the exact measurement and is likely to have a relative error of 12 per cent. During hand-off, when the MH is in the process of receiving data, the HPF variations matter. But as with current technology relating to channel allocation such as the dynamic channel-allocation techniques [12] this has become a matter of negligence.

Dynamic load balancing [13] is done by maintaining two thresholds, viz. Upper Threshold (UT) and Lower Threshold (LT) at the MH. These thresholds are shared by all the computing entities within an MH. This can be achieved by creating the computing entities as threads of the MH. When the load on the MH is greater than UT, a computing entity on that MH leaves the group and increments UT and LT on that MH. Both UT and LT are incremented so that all entities do not leave the groups at the same time. Similarly, when the load on the MH becomes lesser than LT, a computing entity on that MH joins the group and decrements both UT and LT. Two thresholds, UT and LT, are used to avoid oscillations of frequent join and leave.

Further, the load balancing mechanism used is non-pre-emptive [13] and hence migration of an already executing task is not done. This is due to the additional communication overhead involved in process migration.

8.4.3 Parallelism in the Model

The data set which is very large, is multi-cast to the sub-group, on the basis of the size of the file. Multi-cast provides an efficient mechanism for transferring the data to the computing entities for processing. Each computing entity independently splits the task on the basis of its *ID* and *N*. For example, in the case of distributed image rendering application, frames having frame number '*f*' such that mod $(f, N) = ID$ are rendered by the computing entity with identifier ID. When an entity completes its share, it sends the result back to the destination host.

8.4.3.1

A host is assigned as a coordinator to the cluster and it keeps track of the total number of computing entities (called N) under each cluster sub-group. Further, each computing entity has a unique membership identifier (called ID, ranging from 0 to N–1) associated with each group subscribed by it.

8.4.3.2

Whenever an MH wants to participate in distributed processing, the daemon at the MH spawns a set of computing entities, on the basis of its capabilities. The exact number of computing entities spawned by each daemon will depend upon the capability and the processing power of each MH (called HPF, discussed in Section 8.4.2).

8.5 Implementation

The system structure of Moset is shown in Fig. 8.2. A distributed Moset kernel is spread over the nodes that participate in Moset. A Moset kernel consists of multiple local coordinators (*lcs*) to coordinate local activities, multiple co-coordinators (*ccs*) to coordinate the global activities within their cell, and one system coordinator (*sc*) to coordinate the overall global activity of the entire cluster. Each node, either static or mobile, that intends to participate in the Moset kernel runs a local coordinator. MH has a client process and a daemon. The client process and daemon run over a reliable multi-cast protocol. The client processes are used to submit tasks to the MHs (via multi-cast protocol) for distributed processing. The daemons are the computing entities at the MHs that execute a part of the submitted task concurrently with other daemons.

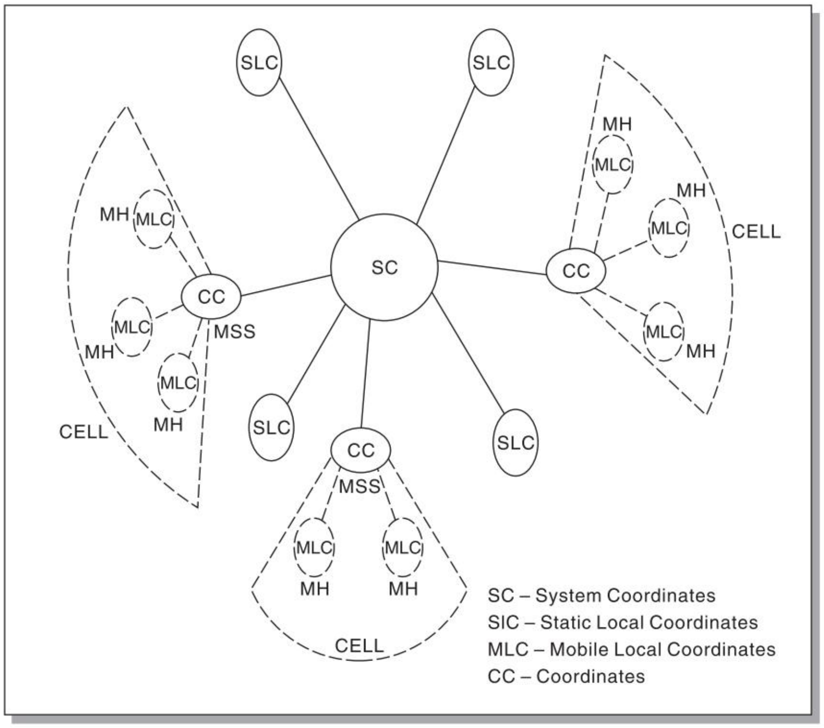


FIG. 8.2

Moset System Structure

8.5.1 Local Coordinator

The *lc* runs on mobile and static hosts that participate in the Moset system, either to improve utilization or to share its work with an anonymous remote node. Any Moset communication to or from the local processes is achieved through the *lc*. In case of image rendering application, the huge data set which is to be rendered, is multi-cast by the *sc* to the sub-group *lcs*, on the basis of the size of the file. Each *lc* independently splits the task on the basis of its *ID* and *N*. For example, frames having frame number '*f*' such that mod $(f, N) = ID$ are rendered by the computing entity with identifier *ID*. When an entity completes its share, the *lc* sends the result back directly to the *sc* if it is executed on a static host or through the *cc* if it is executed on a mobile host.

8.5.2 System Coordinator (*sc*)

The Moset kernel has one *sc* in the logically grouped mobile cluster. The functions of the *sc* are to manage all the nodes that take part in the cluster process and the spawned computing entities, to coordinate all the functions related to task distribution and execution, and to maintain the migration history of the tasks.

Whenever a static machine wants to participate in distributed processing, thus enhancing the machine utilization, it spawns computing entities based on its capabilities through its *lc*. The static machine also includes the MSS which spawns a set of computing entities representing the MHs which are within its cell and which are interested in participating in the computation. As discussed in the previous section, the *sc* groups these computing entities into cluster sub-groups on the basis of the memory capacity of the machine which has spawned the entities. The *sc* keeps track of the total number of computing entities (called *N*) under each cluster sub-group. Further, the *sc* assigns each computing entity a unique membership identifier (called *ID*, ranging from 0 to *N*–1) associated with each group subscribed by it. The exact number of computing entities spawned by each daemon will depend upon the HPF of the node.

The data set which is very large is multi-cast to the sub-groups' *lc*s by the *sc* directly to the static host's *lc*, and through the *ccs* to the mobile host's *lc*, based on the size of the file. Multi-cast provides an efficient mechanism for transferring the data to the computing entities for processing. A history of recent migrations is maintained.

The single point failure of the *sc* is handled by replicating the state of the *sc* in another nearby static node so that in case of failure, the states are not lost and the system can still survive. When an *lc* learns that the *sc* has failed, it initiates the process of identifying the next *sc* by using an election algorithm [14].

8.5.3 Co-coordinator (*cc*)

The Moset has multiple *ccs* running on MSS which has at least one MH participating in the Moset kernel. The *cc* acts as an *sc* with respect to the MHs which are within its cell. Any MH within the coverage area of the MSS, which wants to participate in the sharing of resources, will spawn a set of computing entities to the *cc* running on that MSS. The *cc* collects the set of computing entities spawned and registers with the *sc*. The *cc* takes care of multi-casting the data set to be rendered to the participating MHs and also maintains the history of the execution that takes place in the MH within its cell.

The *cc* also takes care of the mobility of the MHs. When an MH takes the frames for rendering and moves out of the cell and enters another cell, then the new MSS, through hand-off, will be able to inform the *cc* of the old MSS. If the new MSS already has the *cc* daemon running, then it continues with the process by exchanging the information among the *ccs*. In case the new MSS does not run the *cc* daemon, then it gets registered with the *sc* and runs the *cc*.

8.5.4 Time-outs, Mobility and Fault Tolerance

The timeliness issue is an important issue, especially in real-time systems. However, in cluster computing systems, the system is mainly meant for computation-intensive problems like environmental modelling wherein the factor of time can be relaxed. But still the workstation cannot take an infinite amount of time for executing the sub-task which it has accepted to execute. In order to handle this, time-out mechanisms are used. The time-outs are maintained by the *sc* and the *ccs*. A timer is a data counter that ticks at regular intervals. If the workstation does not return within the stipulated time set in the timer, then the sub-task is re-submitted to some other idle workstation for getting executed or in the worst case, it gets executed in the coordinator. In case of sub-tasks executing in MHs, the *cc* takes care of the timer. When a sub-task assigned to an MH does not return before the time-out then the *cc* tries to reassign the sub-task to some other MH within the sub-group, within the cell or as a worst case, it executes the task itself.

The failure of a remote node is detected by the *sc* when the *lc* fails. However, this will work only with static nodes. Mobile nodes may move out of the cell after taking the task for execution and return before the timer time-outs. In this scenario, as the MH was out of coverage for a while, the *cc* will be able to detect this and cannot decide that the MH has failed. Thus the *cc* needs to wait until the timer time-outs. This ensures the fault tolerance of the system.

8.6 Performance

The Moset approach provides parallel programming on a mobile cluster thus improving the utilization of the computing resources of the participating nodes. Moset provides for heterogeneity, fault tolerance and dynamic load balancing to parallel computing. The performance study of the model was done by implementing the distributed image rendering application over the FTEORMP [10]. The FTEORMP is an exactly-once reliable multi-cast protocol. The simulation of FTEORMP was carried out on an object-oriented discrete event simulator in C++ similar to the that used in [15]. The parameters used for simulation are given in Tables 8.1 and 8.2. The cell permanency time in Table 8.2 refers to the average time for which an MH will be inside a cell.

Table 8.1 System Parameters

Parameter	*Value*
Number of Groups	4
Number of MSS'	32
Wired Bandwidth	100 Mbps
Wireless Bandwidth	10 Mbps
Wired Propagation Delay	0.5 msec
Wireless Propagation Delay	0.0 msec
Message Loss Probability	0.001
NACK Loss Probability	0.00
Out-of-range Probability	0.00
NACK Transmission Period	1 second
MCASTREQ Time-out	25 msec
SYNCACK Time-out	500 msec

Table 8.2 Tunable Parameters

Parameter	*Value*
Number of Mobile Hosts	512
Number of Senders	4
Message Rate	250 messages/sec
Message Size	256 bytes
Cell Permanency Time	1 second

Each MH has a client and a daemon. The clients are used to submit tasks to the MHs for distributed processing. The daemons are the computing entities at the MHs, that execute a part of the submitted task concurrently with other daemons. Both client and daemon run over FTEORMP and use the socket abstractions of FTEORMP simulator. The daemons run on the SPARC machines and communicates with the respective MHs of the simulator by sockets. The image rendering application developed renders an image obtained by the CT scan. The characteristic of a CT scan image [16] is that it contains information from a transverse plane only. CT scanners produce three-dimensional stacks of parallel plane images each of which consists of an array of X-ray absorption co-efficients. Due to the availability of stacks of parallel plane images, the volume data sets can be viewed as a three-dimensional field rather than individual planes.

The stack of planes is converted into an image by volume rendering [16, 17] using ray-casting. A ray-casting algorithm casts parallel rays from the viewer into the volume. At each point along the ray, the progressive attenuation due to particle fields is computed. At the same time, the light scattered in the eye direction from the light source is also computed at each point. This rendering procedure is used in the volpack library [18]. The volpack library is an implementation of Reference [17] and it is used in this application.

The task was submitted to render 360 frames of the CT scan data (a frame for each 1° of image rotation). An online image animator written in C, using X11 graphics, [19] was used to animate the frames as and when they arrive from the entities. Due to an online animation, the real-time effects (such as fast rendering when more hosts are involved) were noticed.

The CT scan data of a human head with the skull partially removed to reveal the brain is taken as the input data. The CT scan data consists of 84 slices of 128×128 samples each. The final output frame is 256×256 pixel image. The daemons were run on SPARC 333 MHz and SPARC 500 MHz machines having 256 MB RAM. The daemons use FTEORMP simulator [10] for multi-casting to other MHs. An FTEORMP simulator provides socket abstraction to the applications on the MHs. Hence, the daemons run on the SPARC machines and communicate with the respective MHs of the simulator by sockets.

Table 8.3 shows the results when daemons were executed on SPARC 333 MHz and SPARC 500 MHz under no load condition. On a SPARC 333 MHz, super-linear speed-up is noticed when the number of hosts is two. On a SPARC 500 MHz, super-linear speed-up is noticed when the number of hosts are 2, 3 and 4. The reason for super-linear speed-up is because during the entire computation of 360 frames, the full volume data (which is reasonably large–1.37 MB) needs to be in memory. During this time, page faults occur and the computation time increases. The context switch time is very negligible as the experiment was conducted in no load conditions (when the load on the machines was only due to the daemons). When the task is split into two hosts, the computation time is only for half the task. Hence, the data needs to be in the memory for a much lesser time. As the memory residence time reduces, the page fault overhead also decreases. Due to reduction in these overheads (such as page faults and context switches, if any) and by overlapping computation and communication across hosts, the communication delay seems to be nullified. Thus, super-linear speed-up is achieved. But, as the number of hosts increases to 8 and 16, the communication delay seems to dominate. This is partly because, multi-cast simulator runs on a single host and is not distributed. So, a multi-cast is simulated by multiple unicasts. If multi-cast is done physically (or a distributed simulator is used), then the communication overheads can be reduced further. Scalability limitation is also due to the problem size. If problem size is increased, better speed-up could be attained with more nodes.

Table 8.3 No Load Performance Analysis for Ultra SPARC 333 MHz and SPARC 500 MHz

	SPARC 333 MHz		*SPARC 500 MHz*	
Number of hosts	*Time taken (sec)*	*Speed-up*	*Time taken (sec)*	*Speed-up*
1	434		303	
2	211	2.056	149	2.033
3	149	2.912	102	2.971
4	113	3.841	77	3.935
8	61	7.232	42	7.214
16	47	9.234	33	9.181

Figure 8.3 shows the effect of load on total execution time and speed-up, respectively for rendering 360 frames. The figure shows the execution time and speed-up in the presence of multiple half load processes in the processor run queue. The half load on the processor is achieved by using a program called cpuhog [20]. As the cpuhog loads the processor only half the time, it creates a half-loaded environment. The first graph in Fig. 8.3 shows that the execution time increases as the number of half-loaded processes increases. This is due to the increased context switch time, caused by an increase in the number of processes. The second graph in Fig. 8.3 shows that speed-up is almost constant irrespective of the load. If the number of hosts involved in computation increases, then speed-up seems to increase very marginally with the load on the host. The effects of load on the execution time and the speed-up were taken by using cpuhog with memory of 1 MB.

Apart from loading the processor, cpuhog also hogs the memory resource. The image rendering application needs 10 MB of memory space and if cpuhog occupies more memory, then a lesser amount of memory is left for the image rendering application. By varying the memory used by the cpuhog, the effects on execution time and speed-up of the image rendering application can be seen in Fig. 8.4. These figures show an interesting behaviour, which is explained below.

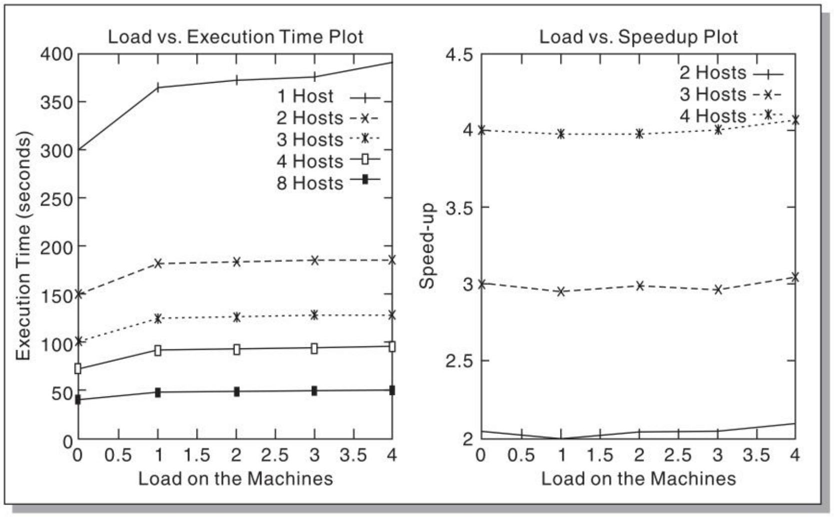


FIG. 8.3

Effect of Load on Execution Time and Speed-up

The first graph in Fig. 8.4 shows that as the memory of cpuhog increases from 1 MB to 200 MB, the execution time for the image rendering application is almost constant with only a very marginal increase. This is because there are not many page faults until the memory occupied by cpuhog increases to threshold. The reason for this behaviour is that due to the availability of large memory space (256 MB), most of the pages are free. But, when the memory occupied by cpuhog increases above a threshold (200 MB), cpuhog encounters frequent page faults. The main reason is that as cpuhog occupies a large number of pages, the oldest page in memory is very likely to be the page belonging to cpuhog. Hence, the oldest page of cpuhog in memory is swapped out and a new page for cpuhog is paged in. As a result, cpuhog sleeps for more time waiting for the required pages to be brought into memory. During the time when cpuhog sleeps, the image rendering application runs. Hence, the total execution time of the application drops sharply as the application gets more time to run.

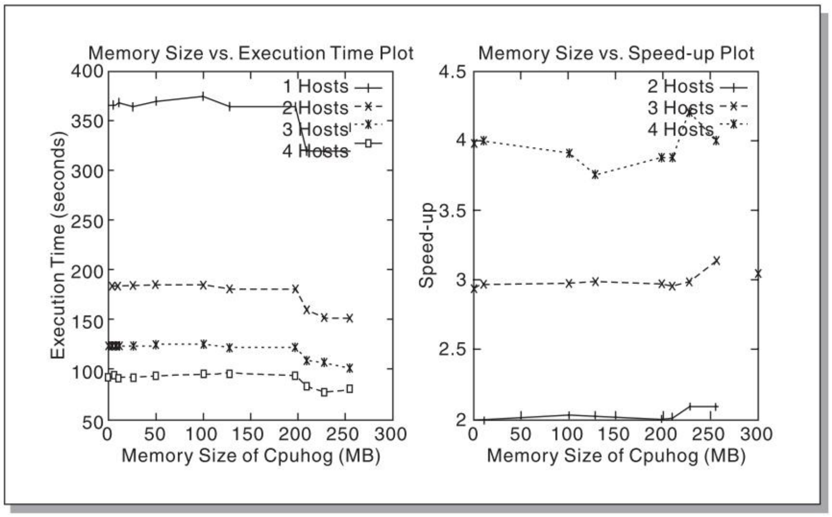


FIG. 8.4

Effect of Process Memory Size on Execution Time and Speed-up

The second graph in Fig. 8.4 shows the effect of memory used by cpuhog, on the speed-up of the image rendering application. The speed-up is almost constant as the memory of cpuhog increases till 200 MB. Upon further increase in the memory usage, the speed-up shows a marginal increase. Moreover, when the application is executed on four hosts, a super-linear speed-up of 4.21 is noticed. Thus, the effect of load on the processor and the memory usage by cpuhog has only a marginal effect on the execution time and on the speed-up of the image rendering application.

8.7 Conclusions and Future Work

This chapter has presented Moset, an ARMCC model based on the integration of mobile and static nodes as clusters interconnected by wireless and wired networks. The model handles the heterogeneity in architecture, operating system, network connectivities and other constraints related to the mobile device. Moset provides one of the first comprehensive efforts at integrating mobile devices into the cluster and effectively harnesses the computing power of mobile devices. Further, mobile devices can also utilize large computing power available in the cluster seamlessly. The performance studies of an image rendering application also demonstrate the feasibility of the idea.

As a future work, the model could be extended to provide inter-sub-task communications. Problems which exhibit a high communication to computation ratio may not be suitable for this model. If, however, the computation time can be overlapped with communication delays, significant speed-ups can be achieved, even in mobile clusters. However, this overlapping is a non-trivial task. Considerable work needs to be done to evolve a generic model for cluster computing and communication for mobile systems. Another interesting direction we are currently pursuing is to extend Moset to a mobile grid, a global collection of resources.

References

1. Litzknow, M., Miron Livny, and Mutka, "Condor–A Hunter of Idle Workstations", Proceedings of the Eighth International Conference on Distributed Computer Systems, pp. 104–111, June 1988.
2. Anderson, T.E., D.E. Culler, D.A. Patterson and the NOW Team, "A Case for Networks of Workstations (NOW)", *IEEE Micro*, Vol. 15, No. 1, pp. 54–64, February 1995.
3. Tandiary, F., S.C. Kothari, A. Dixit and W. Anderson, "Batrun: Utilizing Idle Workstations for Large-scale Computing", *IEEE Parallel and Distributed Technology*, Vol. 4, No. 2, pp. 41–49, Summer 1996.
4. Joshi, Rushikesh K. and D. Janakiram, "Anonymous Remote Computing: A Paradigm for Parallel Programming on Interconnected Workstations", *IEEE Transactions on Software Engineering*, Vol. 25, No. 1, pp. 75–90, January 1999.

5. Johnson, Binu K., R. Karthikeyan and D. Janakiram, "DP: A Paradigm for Anonymous Remote Computation and Communication for Cluster Computing", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12, No. 10, pp. 1052–1065, October 2001.

6. Puder, Arno and Kay Romer, "Internet Users Forecast by Country", *eTForecasts*, *www.Etforecasts.com*, 2003.

7. Zheng, Haihong, Rajkumar Buyya and Sourav Bhattacharya, "Mobile Cluster Computing and Timeliness Issues", *Informatica: An International Journal of Computing and Informatics*, Vol. 23, No. 1, 1999.

8. Cheng, Liang, Ajay Wanchoo, and Ivan Marsic, "Hybrid Cluster Computing with Mobile Objects", Proceedings of the Fourth International Conference on High-performance Computing in the Asia-Pacific Region (HPC-Asia 2000), Beijing, China, pp. 909–914, May 2000.

9. Baist, Abdul and Chin-Chih Chang, "Mobile Cluster Computing Using IPv6", Linux 2002 Symposium, Ottawa, Canada, June 2002.

10. Maluk Mohammad, M.A. and D. Janakiram, "A Fault-tolerant Exactly–once Reliable Multi-cast Protocol for Distributed Mobile Systems", Proceedings of the IASTED International Conference on Communication Systems and Networks (CSN 2003), Malaga, Spain, September 2003.

11. Kiwior, D., J. Kingston and A. Spratt, "PATHMON, A Methodology for Determining Available Bandwidth over an Unknown Network", The MITRE Corporation, Available at *http://www.mitre.org/*, March 2004.

12. Yang, Jianchang, D. Manivannan and Mukesh Singhal, "A Fault-tolerant Dynamic Channel Allocation Scheme for Enhancing QoS in Cellular Networks", Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)–Track 9, pp. 306–315, Big Island, Hawaii, 2003.

13. Singhal, Mukesh and Niranjan. G. Shivaratri, *Advanced Concepts in Operating Systems*, McGraw-Hill Inc., 1994.

14 Garcia-Molina, H. "Elections in Distributed Computing System", *IEEE Transactions on Computers*, Vol. 31, No. 1, pp. 48–59, 1982.

15. Anastasi, Giuseppe, Alberto Bartoli and Francesco Spadoni, "A Reliable Multi-cast Protocol for Distributed Mobile Systems: Design and Evaluation", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 12. No. 10, pp. 1009–1022, October 2001.

16. Watt, Alan and Mark Watt, *Advanced Animation and Rendering Techniques: Theory and Practice*, Addison-Wesley Publishing Company, 1992.

17. Lacroute, Philippe and Mare Levoy, "Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation", *Computer Graphics*, 28 Annual Conference Series, pp. 451–458, 1994.

18. Volpack, "Volpack–A Volume Rendering Library", *www.graphics.standford.edu/software/volpack/*, 1995.

19. Devanathan, V.R. "EMOP: An Exactly-once Multi-cast Protocol for Distributed Mobile Systems, Master's Thesis, Dept. of C.S.E., IIT Madras, Chennai, India, 2002.

20. Libenzi, Davide *CPUHOG–A Kernel Scheduler Latency Tester*, Free Software Foundation, Inc., Boston, MA, USA, 2001.

Chapter 9

Distributed Simulated Annealing Algorithms for Job Shop Scheduling*

9.1 Introduction

Job Shop Scheduling (JSS) belongs to the class of NP–hard optimization prob1em. Even in the NP-hard class of problems, JSS appears to belong to the more difficult ones [1]. The JSS problem can be described as follows. A set of jobs whose operations are to be processed on a set of machines is given. Each job consists of a sequence of operations. Each of these operations has to be processed uninterrupted on a given machine for a specified length of time. There is an additional constraint that each machine can process at most one operation at a time. A schedule is an allocation of the operations to time intervals on the machines. The problem is to find the schedule of minimum time. Solving the JSS problem requires a high computational effort and considerable sophistication. A much simpler task is to find a reasonably good schedule, though not necessarily the optimum one. A number of algorithms have been developed to address this simpler task. They include the Giffler and Thompson algorithm [2], the shifting bottleneck algorithm [3], the simulated annealing (SA) algorithm [1], [4] etc. Computational results show that SA can find shorter makespans than the other recent approximation

*K. Krishna, K. Ganeshan, D. Janakiram

algorithms [1]. This is, however, at the cost of large execution times.

Attempts to reduce the execution time taken by the SA algorithm required that it be distributed. One approach to distribution is to divide the problem space and then distribute the problem space to various nodes in a distributed network. The other approach is to change the temperature modifier parameter. In this approach, the algorithm is run on various nodes of the network on the entire problem space. However, the size of the steps at which the temperature is decreased is raised so that the time taken by the algorithm on each node is less. The effect of raising the temperature modifier is that the probability of the algorithm getting struck at a local minimum is high. However, the probability that two nodes will get struck at the same local minimum is less. Hence the minimum obtained among all the nodes will likely to be the global minimum.

In this chapter, we discuss both these approaches. We also show that depending upon the problem size, one of these approaches can be selected.

9.2 Overview

In this section, a brief overview of the problem, SA and Distributed Algorithms has been provided.

9.2.1 The Problem

The problem can be stated mathematically in the following fashion. Given a set A of n jobs, a set B of m machines and a set C of N operations, $max_{v \in CA} s_v + t_v$ is minimized subject to

$$S_v \geq 0 \text{ for all } v \in C \qquad \text{(equation 1)}$$

$$S_w - S_v \geq t_v \text{ if } v \text{ precedes } w;\ v, w \in C \qquad \text{(equation 2)}$$

$$S_w - S_v \geq t_v V S_v - S_w \geq t_w \text{ if } m_v = m_w, \text{ if } v, w \in C \qquad \text{(equation 3)}$$

where v, w are any operations belonging to the set C

S_v = start, time of the operation v

t_v = processing time of the operation v

m_v = machine on which the operation v is performed.

The following assumptions are made for the problem:

1. All the jobs are available at time zero.
2. There are no machine breakdowns.
3. Operation times of the jobs on the machines are known beforehand.

Table 9.1 4-Job, 4-Machine (4 × 4): Example Problem (EXP1)

Operation number	*Machine number*	*Operating time (sec)*	*Machine number*	*Operating time (sec)*	*Machine number*	*Operating time (sec)*	*Machine number*	*Operating time (sec)*
	Job 1		*Job 2*		*Job 3*		*Job 4*	
1	2	38	2	54	2	37	3	29
2	3	26	3	26	1	24	4	33
3	1	31	4	45	4	23	2	55
4	4	39	1	28	3	34	1	39

A disjunctive graph model (G) with a set of vertices (V), a set of arcs (A) and a set of edges (E) can be used for representing the problem [1]. The disjunctive graph $G = (V, A, E)$ is defined as follows:

The set of vertices V consists of all the vertices in C and two vertices numbered 0 and $N + 1$, representing the fictitious start and end operations respectively. The processing time of the operation is denoted as the weight of the vertex. The two fictitious operations 0 and $N+1$ have operation times of zero.

The set A contains arcs connecting consecutive operations of the same job, as well as arcs from 0 to the first operation of each job and from the last operation of each job to $N + 1$.

The edges in the set E connect operations to be processed by the same machine.

This is illustrated by taking example 1 (refer to Table 9.1). The example problem can be represented as a disjunctive digraph (refer to Fig. 9.1(a)). The set of vertices, arcs and edges in this case are {0, 1, 2, 3, 4...17}, {(0, 1), (0, 5), (0, 9)(0, 13), (1, 2)...} and {(5, 1), (6, 2), (15, 5),…}, respectively.

The directed arcs in Fig. 9.1 denote that the operations are to be processed in that order. The edges in the graph have two possible orientations. For example, if $(v, w) \in E$, then the edge can be directed from either (v, w) or (w, v). The assignment of orientations to these edges forms a schedule. This schedule will indicate the complete sequence of operations to be carried out on each machine. The longest path in this digraph called the 'makespan' gives the total time for completion of the operations of all the jobs. The objective will be to assign orientations to the edges such that the makespan is minimum. The orientations of the edges decide the sequence of operations performed on one machine. This is equivalent to finding the permutations of the operations on each machine to minimize the makespan. This problem is one of the hardest combinatorial optimization problems. In order to solve this problem, a heuristic method called SA is employed.

9.2.2 Simulated Annealing (SA)

SA belongs to the type of local search algorithms [5]. The algorithm chooses an initial solution at random. A neighbour of this solution is then generated by a suitable mechanism and the change in the cost function of the neighbour is calculated. If a reduction in the cost function is obtained, the current solution is replaced by the generated neighbour. On the other hand, if the cost function f of the neighbour is more, the generated neighbour replaces the current solution with an acceptance probability function given by: EXP($-\{f[j] - f[i]\}/T$) where $f[j]$ and $f[i]$ are the cost functions of the generated state and the present state, respectively. T is a control parameter which corresponds to the temperature in the physical annealing process. The above acceptance function implies

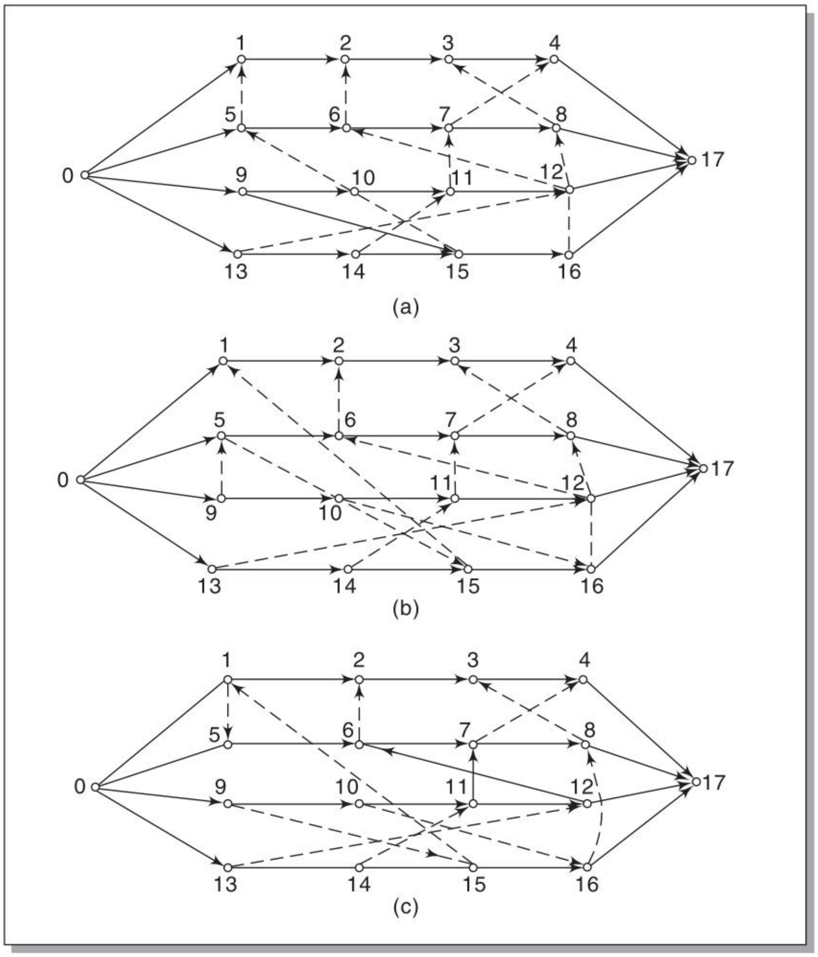


FIG. 9.1

(a) Sub-task 1: Initial Schedule Obtained for EXPI after Locking Edge 15-5, (b) Sub-task 2: Schedule Obtained after Reversing Edge 15-5 from the Initial Schedule (c) Sub-task 3: Schedule Obtained after Removing edge 15-5 from the Initial Schedule

that small increases in f are more likely to be accepted than large increases, and also that when T is high, most of the generated neighbours are accepted. However, as T approaches zero, most of the cost increasing transitions are rejected. The initial temperature in the SA algorithm is kept high such that the algorithm does not get trapped in a local minimum. The algorithm proceeds by generating a certain number of neighbours at each temperature, while the temperature parameter is gradually dropped. This algorithm leads to a near optimal solution.

9.2.3 Distributed Algorithms

Distributed algorithms represent the algorithmic formulation of Distributed Problem Solving (DPS) [6]–[9]. DPS can be termed as co-operative problem solving by a loosely coupled network of problem solvers. The main purpose of distributed algorithms is to exploit the processing power of a number of nodes on a network. Since the SA technique for JSS is inherently sequential and highly compute-intensive, distributed algorithms for the SA technique for JSS can make the technique applicable for large-scale problems. Two different approaches to the development of distributed algorithms for SA technique for JSS have been contemplated. One approach is to divide the problem space into independent sub-tasks. The algorithm based on this approach is termed as the Locking Edge algorithm. This algorithm is modified for large problem sizes. The other approach involves distributing the reduction rate of the temperature among various nodes of the network. The algorithm based on this approach is called the Temperature Modifier algorithm. These algorithms have been explained in the subsequent sections.

9.3 Distributed Algorithms for Job Shop Scheduling

This section describes the development of distributed algorithms for JSS, using the SA technique. Initially a sequential algorithm [1], [5] is presented which will be modified subsequently to develop distributed algorithms.

9.3.1 Sequential Algorithm

The sequential algorithm involves the following major steps:

1. Finding all initial schedules,
2. Evaluating cost of the schedule,
3. Finding the critical path,
4. Generating a neighbour. These are discussed in detail below.

9.3.1.1 *Initial Schedule*

Given a disjunctive graph $G = (V, A, E)$ for solving the problem, an initial schedule is generated. The Giffler and Thompson algorithm [2] is employed for this purpose. The algorithm attempts to construct the schedule by considering all the operations (n) on all the machine (m), with the criteria employed being the earliest starting time and the processing time of each of the operations. At each stage an operation not yet included in the schedule and requiring a minimum time is chosen and included in the partial schedule. The partial schedule becomes a complete schedule when all the operations of the jobs are included in the schedule. The generated schedule can be represented as a digraph.

9.3.1.2 *Cost Function*

After obtaining the digraph representing all initial schedules, the earliest and the latest start times of each of the operations in the graph are calculated. The Critical Path Method (CPM) is used for this purpose. The makespan is the earliest start time or the latest start time of the last operation. This forms the cost of the schedule.

9.3.1.3 *Critical Path*

After evaluating the cost function, the critical path in the digraph is identified. The critical path can be defined as a set of edges from the first vertex to the last vertex which satisfy the following properties:

(a) The latest start time and the earliest start time of each vertex on the edge must be the same.

(b) For the same edge $u \rightarrow v$, the sum of the start time and the operation time of u must be equal to the start time of v.

An edge in the critical path is reversed to generate a neighbour and this is discussed in the next section.

9.3.1.4 *Generating a Neighbour*

The neighbourhood of a schedule can be defined as a set of schedules that can be obtained by applying the transition function on the given schedule. Neighbourhoods are usually considered by first choosing a simple transition function. A transition in the case of a JSS problem is generated by choosing the vertices v and w (as given in [1]). The following facts need to be considered.

(a) v and w are any two successive operations performed on the same machine k;

(b) $(v, w) \in E_i$ is a critical edge, i.e (v, w) is on the longest path of the digraph.

A neighbour is generated by reversing the order in which v and w are processed on the machine k. It has been shown that by using this transition function, it will be possible to eliminate infeasible solutions and also keep non-decreasing paths out of the search space [1].

Thus, in the digraph such a transition results in reversing the edge connecting v and w and replacing the edges (u, v) and (w, x) by (u, w) and (v, x) respectively, wherein u is the previous operation to v on the same machine, and x is the next operation to w on the same machine.

9.3.2 Distributed Algorithms

We have developed three distributed algorithms for JSS by modifying the sequential algorithm. They can be termed as:

1. Temperature Modifier Algorithm,
2. Locking Edges Algorithm, and
3. Modified Locking Edges Algorithm.

They are explained in detail below.

9.3.2.1 *Temperature Modifier Algorithm (TMA)*

The choice of the temperature modifier in the sequential algorithm affects the probability of the algorithm getting struck at a local minimum. A low value for the modifier makes the algorithm fast but the probability of the algorithm getting struck at a local minimum is high. In order to reduce this probability, the sequential algorithm can be simultaneously executed on different nodes. The Temperature Modifier Algorithm is briefly stated in Table 9.2.

9.3.2.2 *Locking Edges Algorithm (LEA)*

It can be observed that the TMA does not result in much improvement with respect to the execution time on the computer. The LEA has been developed to improve this. This algorithm generates sub-tasks by 'locking' edges in the digraph. The term 'locking' can be defined as marking an edge of the digraph such that its orientation cannot be changed. If the number of edges locked in the digraph is m then we can generate 3^m equal sub-tasks. This is equivalent to dividing the entire search space into 3^m divisions. For example, if the number of locked edges in the digraph is one, then we can generate three sub-tasks. These sub-tasks can be obtained as follows:

(a) In the first sub-task, the search space consists of all possible orientations of the edge except that of the locked edge. The orientation of the locked edge remains intact.

(b) In the second sub-task, the orientation of the locked edge is reversed.

(c) In the third sub-task, the locked edge is removed.

The generation of three sub-tasks is illustrated by taking example 1 (please refer to Table 9.1). The initial schedule generated for the example problem is represented in Fig. 9.1(a). In this schedule, the edge 15–5 is locked to generate the sub-tasks. Hence the initial schedule in Fig. 9.1(a) with edge 15–5 locked forms the starting schedule for sub-task 1. The starting schedule for the

Table 9.2 Temperature Modifier Algorithm

```
1. Select appropriate temperature modifiers for different nodes.
2. Run the following sequential algorithm with the chosen
   temperature modifier on each node.
   (a) Generate the initial schedule, given the processing times or
       all operations and the machine order for each job.
   (b) Repeat
       Counter = 0.
       Compute the cost of the initial schedule [t[i]];
       Repeat
       Calculate the critical path;
       Generate the neighbourhood;
       Compute the cost of the Generated schedule [t[j]];
       Accept or reject the generated schedule with the probability
       min(1.exp (–{t[j] – t[i]}/T));
       until (counter = number_of_rune_at_a_temperature);
       T = T + t_modifier;
       until (T = 0 V Minimum schedule not changed for a long
       time);
3. Send the result back to the central node.
```

second sub-task is generated by reversing the locked edge 15–5 in the initial schedule and the schedule thus obtained is given in Fig. 9.1(b). The starting schedule for the sub-task 3 is generated by removing the locked edge 15–5 from the initial schedule and this is shown in Fig. 9.1(c).

The above concept can be further extended to a case in which there can be *m* locked edges. Figure 9.2(a)–(c) explains the method of generation of the sub-tasks. The generated sub-tasks are assigned to the cooperating nodes in the Distributed Problem Solving (DPS) network. The detailed algorithm is presented in Table 9.3.

The edges to be selected for locking are chosen at random. The edge chosen affects the division of the search space and influences the quality of the solution in some cases. This is discussed in the next section.

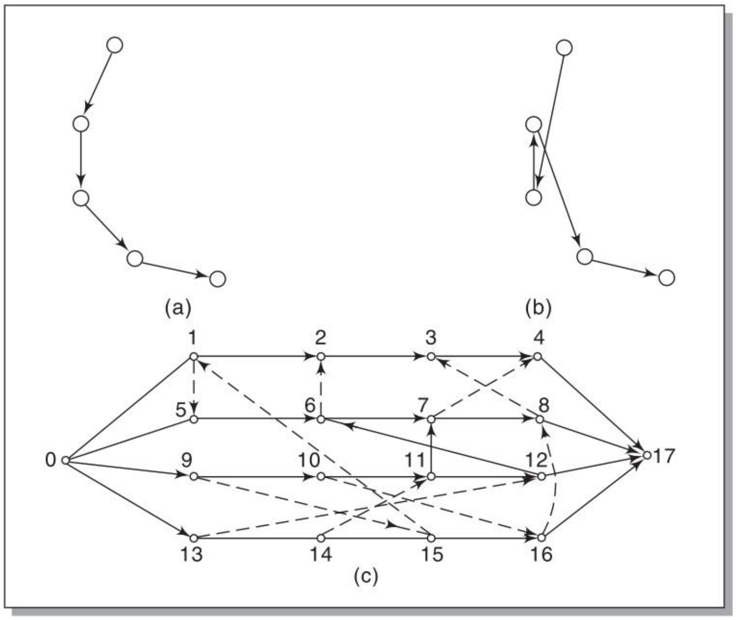


FIG. 9.2

Sub-task Generation: (a) EDGE → Intact (b) EDGE → Reversed (c) EDGE → Removed

Table 9.3 Locking Edges Algorithm

1.	Generate the initial schedule with the input given.
2.	Choose the number of edges to be locked, say *m*.
3.	Generate the sub-tasks depending upon the number of locked edges i.e. for *m* locked edges 3 power *m* sub-tasks are generated.
4.	Assign each of the sub-tasks to the cooperating nodes, pass information about the locked edges.
5.	Wait for the results from the cooperating nodes.
6.	Choose the optimal cost solution.

9.3.2.3 *Modified Locking Edges Algorithm (MLEA)*

It is observed that as more and more edges are locked, it results in a decrease in the performance with respect to the optimal scheduling cost in some cases. This is explained theoretically in the subsequent paragraphs.

A new term called 'collision' is defined in the case of the locking edges version of the distributed algorithm. In the locking edges version, the search space is divided equally among all the co-operating nodes. For a three-node distribution, the search space can be represented as in Fig. 9.3(a) and for a nine-node version, it can be represented as in Fig. 9.3(b). One of the edges on the critical path is chosen at random for generating a neighbour. If this selected edge (say *e*1) affects the locked edge (say *e*2), then such a situation is termed as 'collision of edge *e*1 with locked edge *e*2'. In such cases, the edge *e*1 is rejected and a new edge is selected at random to generate a neighbour. It can be interpreted as a node trying to 'penetrate' into the search space belonging to another node. It can be observed from the figures that these collisions will be more in the nine-node case compared to that in the three-node case. This results in a marginal increase in the schedule cost as compared to the previous case. In cases where the solution corresponding to the minimum exists on the boundary or in the search spaces of two nodes, there is less likelihood of it being reached in the nine-node case because of collisions.

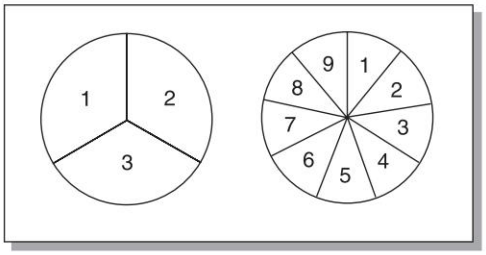


FIG. 9.3

Search Space Division: (a) Three-node Distribution (b) Nine-node Distribution

In order to minimize the affect of collisions, a modification to the 'locking edges' version is attempted. This can be explained as follows. Consider a case wherein a single edge is locked. Here there are two sets of nodes. The first set of nodes consists of three nodes and will be assigned the same tasks as in the 'locking edges' case. The second set of nodes, also consisting of three nodes, are assigned tasks in such a way that the boundary points of the previous set of search spaces become the active search spaces for these nodes. As pointed out previously, this is done mainly to reduce the effect of collisions. This can be extended to any number of locked edges. The detailed algorithm is given in Table 9.4.

Table 9.4 Modified Locking Edges Algorithm

1.	Generate the initial schedule with the input given.
2.	Choose the number of edges to be locked, say *m*.
3.	Generate the first set of sub-tasks (3 power *m*).
4.	Assign the sub-tasks to the cooperating nodes. Pass information about the locked edges also.
5.	Generate the second set of sub-tasks (3 power *m*).
6.	Assign the sub-tasks to the cooperating nodes. Pass information about the locked edges also.
7.	Wait for the results from the cooperating nodes.
8.	Choose the optimal cost solution.

The edges to be locked are selected at random and the selected edges may be the cause for collisions. However, it is not easy to predict beforehand this effect of a particular edge on collisions. Hence, in the absence of such knowledge, the modified locking edge version guarantees that the search space is divided to avoid excessive collisions at least in one set of the search space division.

9.4 Implementation

These algorithms have been implemented on the Distributed Task Sharing System (DTSS) [7] running on a network of Sun Workstations having three servers of Sun 3/60 and 15 clients or Sun 3/50 connected together by a thin Ethernet. The DTSS has been developed around a message kernel. The message kernel is implemented by using datagram sockets. Messages across nodes are transferred by these datagrams. The message kernel provides support for reconfiguring the nodes on the network, and for sending and receiving the task award and the result messages. The nodes on the network are initially configured such that one of the nodes is identified as a central node. During the initial configuration, many other required client nodes are also identified. The central node has the responsibility of dividing the search space and of communicating the tasks to the client nodes through task award messages. After receiving the task award messages, the client nodes execute the required task and send back the results through the result messages. The network is highly flexible and can be reconfigured with any number of client nodes.

The messages are retransmitted in the case of loss of message packets in transmission. This is detected by the non-receipt of an acknowledgement for the packets sent within a specified time-out period. The node failures are also detected in a similar fashion. In case of node failures, the corresponding sub-tasks are assigned to other client nodes available on the network.

Before proceeding further on the implementation details, two terms are defined:

- **Central Node:** This node holds the responsibility of task division, task award to client nodes, receiving result messages and synthesis of the final solution from the results obtained from the client nodes.
- **Client Node:** This node solves the sub-task assigned to it and returns the result.

9.4.1 Temperature Modifier Algorithm (TMA)

The implementation of the algorithms has been carried out by a central node and a set of client nodes. The central node generates the initial schedule, given the processing times of all the operations and the machine order for each job. This node then sends the initial schedule and also different temperature modifier parameters to each of the client nodes on the network. The client nodes execute the sequential algorithm (Table 9.2) with their corresponding temperature modifiers and send back the result to the central node.

After receiving the results from the client nodes, the central node chooses the solution that has the minimal cost.

The quality of the solution generated is influenced by the cooling rate. We employ a three parameter cooling schedule, as seen in [1]. The parameter delta controls the rate of cooling. A lower value of delta reflects a slower cooling rate and consequently the algorithm takes more time. The value of delta employed is in the range of 10^{-1}–10^{-4}. Hence, it is important to choose an appropriate value of delta for each node in the network. A marginally high value for delta than that employed in the sequential algorithm can be chosen for TMA. For example, if a delta value of 10^{-2} is employed for a sequential algorithm, for a TMA case, a delta of 0.5×10^{-1} may be chosen. As a higher cooling rate affects the quality of the solution, it may be appropriate to employ TMA only in small size problems, where LEA or MLEA doesn't give better results.

9.4.2 Locking Edges Algorithm (LEA)

As in the above implementation, here too a central node and a set of client nodes participate in the execution of the problem. In this case, the number of client nodes is equal to the number of generated sub-tasks. The generated sub-tasks are based on the number of locked edges.

Initially, the central node generates the initial schedule and depending upon the number of locked edges, it generates the subtasks and assigns each of the latter to one of the client nodes. The client nodes execute the sub-task and return the result to the central node. The central node synthesizes all the results and chooses the minimum cost solution as the best one.

9.4.3 Modified Locking Edges Algorithm (MLEA)

In order to implement this algorithm, a central node and sets of client nodes are identified. The central node divides the search space into sub-tasks and assigns them to one set of client node. This process is same as the one described in Locking Edges algorithm. However, in the case of the MLEA, the search space is again divided by the central node such that the boundaries of the search space in the earlier set become the active search spaces in this case. The sub-tasks generated in this process are assigned by the central node to the next set of client nodes. Thus many sets of client nodes participate in problem solving in this case.

9.5 Results and Observation

9.5.1 Comparison of Results of Sequential Implementation

We attempted a comparison of the results of our implementation of the sequential algorithm with that of the Lar Vaanhoven, Aarts, and Lenstra (VAL) [1] implementation. We have considered the same three problem instances, FIS1, FIS2 and FIS3 from Fischer and Thompson [10]. FIS2 is one of the difficult test cases and it is stated that it defied solution to optimality for more than twenty years [1]. It has been shown that Sequential Simulated Annealing (SSA) produces better results compared to the Adams, Balas and Zawack (ABZ) method [3] and the Matsuo, Suh and Sullivan (MSS) method [4] but this is at the cost of large running times [1].

Table 9.5 compares the results of our implementation with that of the VAL implementation. We have considered the three parameter cooling schedule as in the VAL implementation. The parameter delta controls the cooling rate. A lower value for delta means a lower cooling rate. The results of our sequential implementation are marginally better than the VAL implementation. Our observations regarding sequential implementation of the algorithm are as follows:

1. The initial and final temperatures for different problem sizes cannot be the same. We found that for higher problem sizes, it may be advantageous to keep the initial temperature high as the search space is higher in this case.
2. The number of neighbours generated at any temperature also varies depending upon the problem size. However, the bound $m \times n - m$. (m is the number of machines and n is the number of jobs) on the neighbours generated at a temperature appears inappropriate for small size problems. This bound is not able to generate a markov chain of sufficient length at a given temperature for convergence to the global optimum.

Table 9.5 Comparison of the Results of Sequential Implementation

Problem size	*Delta*	*Results of the sequential implementation cost (ave)*	*Results of val implementation cost (ave)*
6 + 6	0.1	55	56
(FIS1)	0.01	55	55
10 + 10	0.1	1027	1039.6
(FIS2)	0.01	985	985.8
20 + 5	0.1	1267.5	1354.2
(FIS3)	0.01	1227	1229.0

9.5.2 Discussion of Implementation Results of the Distributed Algorithms

As stated earlier, one of the major drawbacks of the sequential SA algorithm is the large execution times taken by it. The main purpose of developing the distributed algorithms is to reduce the execution times.

We compare the distributed algorithms with the sequential algorithms based on the improvement in the execution times and the quality of the solution obtained.

We have taken several instances of the JSS problem by varying the size of number of jobs and the number of machines. These instances of the problem are generated by using an algorithm which assigns operation times from 1–90 to the operations at random.

The performance of the various distributed algorithms for these problem instances is tabulated in Table 9.6. The three problem instances from Fischer and Thompson have also been implemented and the performance of these problem instances has been tabulated in Table 9.7. Graph 1 (Fig. 9.4) shows the plot of problem size versus execution time and Graph 2 (Fig. 9.5) shows the plot between the problem size and the percentage deviation from the optimal solution cost.

Following is a summary of the results of the distributed algorithms:

1. Distributed algorithms generally performed well from the viewpoint of execution time. LEA performed extremely well and gave a near linear speed-up.
2. In the case of LEA, with an increase in the number of edges locked, the quality of the solution comes down considerably. This is explained by an increase in the number of collisions with more edges locked. When a collision occurs, that part of the search space is ignored by the node. Hence, the probability of searching some part of the solution space is reduced.
3. The MLEA performs better as compared to the pure locking edge algorithm from the viewpoint of the quality of solution. This is mainly due to the creation of overlapping search spaces and allocating those search spaces more nodes on the network. This makes the probability of searching the various solution spaces equally high.

Table 9.6 Implementation Results

	SSA		*TMA*		*LEA*			*MLEA*	
Problem size	*C*	*T*	*C*	*T*	*C*	*T*	*n*	*C*	*T*
(10 + 5)	837	224	812	270	826	63	4	826	87
					812	58	10		
(5 + 10)	890	102	890	102	890	33	4	890	33
(15 + 5)	1084	522	1084	516	1084	419	4	1084	456
					1111	99	10		
(10 + 10)	966	577	896	1146	915	703	4	915	791
					950	149	10		
(10 + 10)	964	1788	954	1845	959	692	4	959	693
					1029	258	10		
(20 + 5)	1352	1160	1338	1152	1338	468	4	1338	587
					1338	316	10		
(25 + 10)	1603	1402	1548	2070	1601	877	4	1601	869
					1639	293	10		
(10 + 17)	1540	2864	1535	2840	1589	661	4	1554	1330
					1595	647	10		
(20 + 10)	3309	7439	3309	7340	3309	2961	4	3309	2916
					3449	1133	10		
(10 + 20)	2000	2200	2000	2544	2000	1526	4	2000	2379
					2052	686	10		
(15 + 20)	3391	6701	3391	6666	3391	2221	4	3391	2227
					3391	1018	10		

Legend:
SSA: Sequential Algorithm
TMA: Temperature Modifier Algorithm
LEA: Locking Edges Algorithm
MLEA: Modified Locking Edges Algorithm
C: Cost of best solution obtained
T: Execution time (secs)
n: number of co-operating nodes
Problem size: Number of jobs + Number of machines

Table 9.7 Comparison of Implementation Results

		SSA		*TMA*		*LEA*		*MLEA*		O_C
P	*D*	*C*	*T*	*C*	*T*	*C*	*T*	*C*	*T*	
6 + 6	1	55	117.0	55	122.56	56	72.06	56	86.81	
(FIS1)	2	55	506.3	–	–	56	85.18	55	123.2	55
10 + 10	1	1027	1400	1027	2000	1006	487	1002	802.1	
(FIS2)	2	903	16200	903	18000	983.3	2069	983.3	4325	930
20 + 5	2	1267.5	1950	1247	1290.7	1256.6	254.6	1234	1345.7	
(FIS3)	2	1227	18600	1227	18200	1217.5	7711.6	1214	5698	1165

Legend:
SSA: Sequential Algorithm
TMA: Temperature Modifier Algorithm
LEA: Locking Edges Algorithm
MLEA: Modifier Locking Edges Algorithm
P: Problem size (Number of jobs + Number of machines)
D1: Value of delta: 0.1
D2: Value of delta: 0.01
C: Average cost of the solution obtained
T: Execution time (secs)
O_C: Optimal cost of the solution

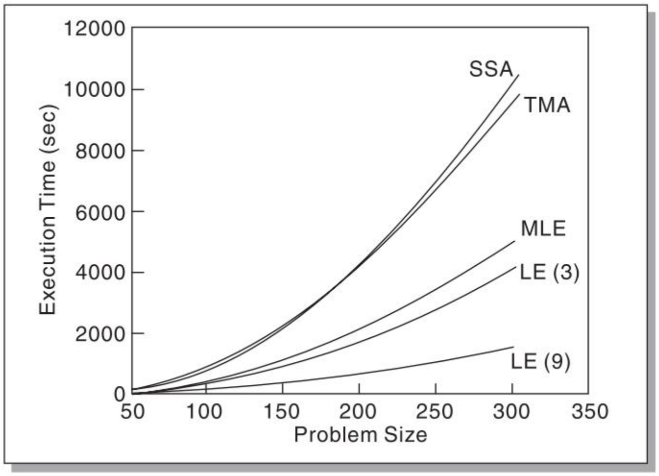


FIG. 9.4

Graph 1–Plot Showing Execution Performance

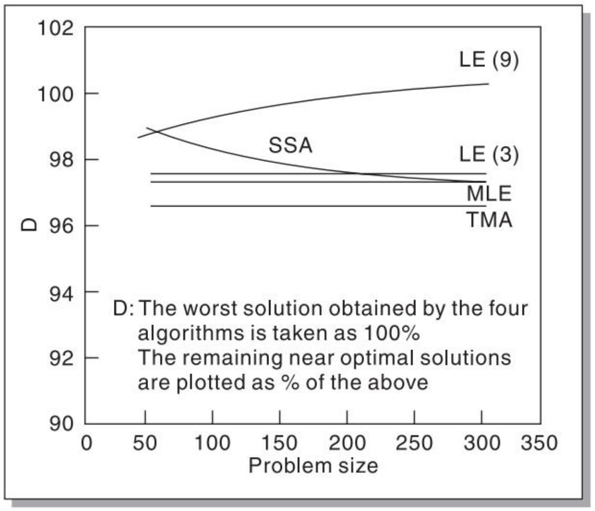


FIG. 9.5

Graph 2–Plot Showing the Solution Cost Performance

4. The performance of the MLEA with respect to the execution time is comparable with that of the LEA. However, twice the number of nodes are employed in the MLEA as compared to the LEA.

5. The TMA is the simplest of the three algorithms and does not involve any problem division overhead. Hence it is better suited to small size problems wherein the search space is small. If the search space is small and if it is further subdivided, it may lead to nodes getting into the search space or neighbouring nodes often. This may lead to excessive collisions. Hence, the TMA is ideal in cases when the search space is small.

In conclusion, we observe that the TMA is the best suited for problem in case of sizes less than 10×10, the MLEA in case of problem sizes between 10×10 and 20×20, and the LEA in case of sizes above 20×20.

9.6 Conclusions

Different distributed algorithms have been implemented and their performance measured. The study shows that in case of very large problem sizes, the LEA give better performance. For medium sized problems, the TMA and MLEA give better performance. Tests also show that the distributed algorithms perform better as compared to the sequential algorithm with respect to the optimal solution cost and the execution time.

References

1. Van Laarhoven, P.J.M., E.H.L. Aarts and J.K. Lenstra, "Job Shop Scheduling by Simulated Annealing", *Operation Research*, Vol. 40, pp. 113–125, 1992.
2. French, S., *Sequencing and Scheduling: An Introduction to the Mathematics of Job Shop*, Chichester, UK Horwood, 1982.
3. Adams, J., E. Balas and D. Zawack, "The Shifting Bottleneck Procedure for Job Shop Scheduling", *Mgmt. Science*, Vol. 34, pp. 391–401,1988.
4. Matsuo, H., C.J. Suh, and R.S. Sullivan, "A Controlled Search Simulated Annealing Method for the General Job Shop Scheduling Problem," Working Paper 03–04-88, Department of Management, University of Texas, Anstin, 1988.
5. Eglese, R.W., "Simulated Annealing: A Tool for Operation Research", *Euro. J. Operation Res.*, Vol. 46, pp. 271–281, 1990.
6. Durfee, E.H., V.R. Lesser and D.D. Corkjll, "Trends in Co-operative Distributed Problem Solving," *IEEE Trans. Knowl. Data Eng.*, Vol. 1, No. 1, pp. 63–82, March 1989.
7. Janaki, R.D. and K. Ganeshan, "DTSS: A System for Implementing and Analysing Distributed Algorithms", Indian Institute of Technology, Madras, Tech. Rep. IITM-CSE-93-00l, January 1993.

8. Allwright, J.R.A. and D.B. Carpenter, "A Distributed Implementation of Simulated Annealing for the Travelling Salesman Prob1em", *Parallel Computing*, Vol. 10, pp. 335–338, 1989.

9. Decker, K.S., "Distributed Problem Solving Techniques: A Survey", *IEEE Trans. Syst. Man Cyber.*, Vol. 17, No. 1, pp. 729–739, 1987.

10. Fisher, H. and G.L. Thompson, "Probabilistic Learning Combinations of Local Job Shop Scheduling Rules," *Industrial Scheduling*, J.F. Muth and G.L. Thompson, (eds), Englewood Cliffs, Prentice Hall, New Jersey, pp. 225–251, 1963.

Chapter 10

Parallel Simulated Annealing Algorithms*

10.1 Introduction

Simulated Annealing (SA) has been considered a good tool for complex non-linear optimization problems [5, 2]. This technique has been widely applied to a variety of problems. One of the major drawbacks of the technique is its very slow convergence.

There have been many attempts to develop parallel versions of the algorithm. Some parallel systems exist for achieving close to ideal speed-up on small processor arrays [6]. There are also special purpose architectures for implementing the annealing algorithm [9]. Two different approaches to parallelization of SA exist in literature—single-trial parallelism and multiple-trial parallelism [5]. Very often these approaches are highly problem-dependent and the achievable speed-up depends upon the problem characteristics. In all these cases, there is a need to divide the problem into sub-problems and to subsequently distribute these problems among the nodes or processors.

In this chapter, we present two distributed algorithms for simulated annealing. The first algorithm is named the Clustering Algorithm (CA) and the second algorithm, the Genetic Clustering Algorithm (GCA). In CA, a cluster of nodes works on the search space of the SA algorithm. The nodes in the cluster assist each

*D. Janakiram, T.H. Sreenivas, K. Ganapathy Subramaniam other by exchanging their partial results and this helps the nodes to converge to a good initial solution to start with. In the second algorithm, the genetic algorithm is applied to find the initial n good solutions required as the starting point for the SA algorithm on the n different nodes of the network. The two algorithms have been applied to the JSS and the TSP. Both algorithms showed very good performance in terms of solution time and solution quality.

The rest of the chapter is organized in the following fashion. Section 10.2 describes the SA technique. Section 10.3 presents the CA. Section 10.4 presents the GCA. Section 10.5 gives implementations of both the algorithms on a network of Sun workstations. Section 10.6 deals with the application and performance studies of the algorithms with respect to the cases of JSS and TSP.

10.2 Simulated Annealing (SA) Technique

Often the solution space of an optimization problem has many local minima. A simple local search algorithm proceeds by choosing a random initial solution and generating a neighbour from that solution. The neighbouring solution is accepted if it is a cost-decreasing transition. Such a simple algorithm has the drawback of often converging to a local minimum. The SA algorithm, though by itself a local search algorithm, avoids getting trapped in a local minimum by also accepting cost-increasing neighbours with some probability. In SA, first an initial solution is randomly generated, and a neighbour is found and accepted with a probability of min $(1, \exp(2d/T))$, where d is the cost difference and T is the control parameter corresponding to the temperature of the physical analogy and will be called temperature. On slow reduction of temperature, the algorithm converges to the global minimum, but the time taken increases drastically.

SA is inherently sequential and hence very slow for problems with large search spaces. Several attempts have been made to speed up this process, such as development of parallel SA techniques and special purpose computer architectures.

10.2.1 Parallel Versions of SA

Parallelism in SA can be broadly classified into two approaches– single-trial parallelism and multiple-trial parallelism [5]. But these methods are highly problem-dependent and the speed-up achieved depends wholly on the problem at hand. Another taxonomy divides parallel annealing techniques into the following three major classes:

1. serial-like algorithms,
2. altered generated algorithms, and
3. asynchronous algorithms [1].

Each class of the algorithm makes some trade-off among cost function accuracy, state generation, parallelism, and communication overhead. High-performance special purpose architectures show the promise of solving computationally expensive applications without expensive supercomputers and include specially designed computer architectures to suit the annealing algorithm [11].

10.3 Clustering Algorithm for Simulated Annealing (SA)

Experiments on the SA technique have shown that a good initial solution results in faster convergence. Similar observations have been made in [13]. The proposed distributed algorithms take advantage of this observation. Initially, the n nodes of the network run the SA algorithm by using different initial solutions. After a fixed number of iterations, they exchange their partial results to get the best one. All the nodes accept the best partial solution and start applying the SA technique for that best partial result. They again exchange their partial results after some fixed number of iterations. After repeating this process for a pre-defined number of times, each node works independently on its partial result. The complete algorithm is given in Table 10.1.

Table 10.1 Cluster Algorithm (CA) for Simulated Annealing (SA)

```
Input to the algorithm:
        $n$ = Number of the nodes in the network.
        $p$ = Exchange parameter for partial results.
        $r$ = Reduction parameter for the number of iterations
              before exchange of partial results.
        $i$ = Input graph for scheduling.
Coordinator node algorithm:
1. Distribute the $n$ random initial solutions to the $n$ nodes and wait.
2. Upon receiving the first converged result from any of the nodes,
   stop SA on other nodes.
Worker node algorithm:
1. Accept initial solutions from the coordinator.
2. repeat
      2.1. Execute SA for $p$ iterations. Exchange partial results among
           the worker nodes. Accept the best partial result.
      2.2. $p = p - r^*$ (loop iteration number).
           until ($p = 0$).
3. Execute SA by using the best solution found as the initial solution.
4. Send the converged value to the coordinator.
```

10.4 Combination of Genetic Algorithm and Simulated Annealing (SA) Algorithm

Experiments have shown that a good initial solution for SA improves both the quality of the solution as also the execution time. Genetic algorithms try to improve a set of 208 Ram, Sreenivas, and Subramaniam solutions rather than a single solution. Since we require n initial solutions for distributing among n nodes, we choose to combine SA with GA.

10.4.1 Genetic Algorithm

In GA [11], an initial population consisting of a set of solutions is chosen and then the solutions are evaluated. Relatively more effective solutions are selected to have more offsprings, which are, in some way, related to the original solutions. If the genetic operator is chosen properly, the final population will have better solutions. GA improves the whole population. SA aims at producing one best solution. For the distributed SA implementation, we require several good initial solutions to ensure the fast convergence of SA. We chose GA for obtaining the required number of good initial solutions. The operator used for generating offsprings in JSS is related to the processing order of jobs on the different machines of the two parent solutions. Let PO11, PO12, ..., PO1*m* be the processing orders of jobs on machines 1, 2, ..., *m* in parent1 and PO21, PO22, …, PO2*m* be the processing order on machines 1, 2, ..., *m* in parent2. If random (1, *m*) = *i*, then processing orders in child1 and child2 are PO11, ..., PO1*i*, PO2*i* 1 1, ..., PO2*m* and PO21, ..., PO2*i*, PO1*i* 1 1, ..., PO1*m* respectively. After getting the offspring, a check is made to see if there are any cycles in the offsprings and if there is one, the operation is performed once again by generating another random number. A cycle in a state indicates an invalid schedule. The pseudo-code for the GCA is given in Table 10.2.

Table 10.2 Genetic Clustering Algorithm (GCA)

```
1. Central node generates $n$ initial solutions using GA. It runs GA
   for fixed number of iterations, $t$.
     1.1 Choose initial population of fixed size and set $i = 1$.
     1.2 while ($i <= t$)
         begin
           1.2.1 Apply the operator on the two parent schedules
                 chosen randomly to produce two offspring and
                 replace the parents by the best two out of the four
                 schedules.
           1.2.2 $i = i + 1$
                 end
2. Central node sends $n$ best solutions chosen to the $n$ remote
   worker nodes.
3. Each worker node runs the SA algorithm by using the initial
   state received.
4. Upon receiving a converged result from one of the worker nodes,
   the central node stops execution.
```

10.5 Implementation of the Algorithms

Both the above algorithms have been implemented by using a platform called DiPS (Distributed Problem Solver) [3] running on a network of 18 Sun workstations. It is built on a communication kernel. Using the kernel, it is possible to send task award messages, task result messages, configure messages, and partial result messages, among the various nodes of the DiPS network. The full implementation details of both algorithms are given in the subsequent sections.

10.5.1 Implementation of the Clustering Algorithm (CA)

In the CA, the central node executes the code in Table 10.3 and the worker nodes the code in Table 10.4. The algorithm for SA is given in Table 10.5.

Table 10.3 Clustering Algorithm for the Central Node

```
1. Initialize ( ).
2. Generate $n$ random initial states and assign to the $n$ nodes of the
   network.
3. Wait for results.
4. Output_Results.
```

Table 10.4 Clustering Algorithm (CA) for the Worker Node

```
1. Get the sub-task from the central node and $p$, the exchange
   parameter.
2. **while** ($p > 0$)
   **begin**
      2.1 Simulated_annealing ($n$).
      2.2 Send the best solution obtained to the central node.
      2.3 $p = p -$ (loop_iteration_value)$^8$ $r$.
          **end**
3. Run SA.
4. Send the converged value to the central node.
```

Table 10.5 Simulated Annealing

```
Simulated annealing (n)
  begin
 1. Set t = Initial_temperature
 2. repeat
        2.1 Counter = 0.
        2.2 repeat
              2.2.1 Compute the cost of the schedule ($f[i]$).
              2.2.2 Find the critical path schedule.
              2.2.3 Generate a neighbour and compute the cost of the
                    neigbour ($f[j]$).
              2.2.4 Accept or reject the neighbour with a probability
                    of min(1, $e^{-(f[i] - f(l)/t)}$.
              2.2.5 Increment counter.
                    until (Counter = Number of iterations at t).
 3. t = t* temp_modifier.
 4. After every n iterations exchnage results and accept the best
    schedule found.
    until (shopping criteria)
     end.
```

10.5.2 Implementation of Genetic Clustering Algorithm (GCA)

In the case of GCA, first the genetic algorithm is run on the central node to get the required *n* initial solutions. These initial solutions are used by the *n* client nodes of the distributed systems as a starting solution for the SA algorithm. The code that is executed on the central node is the same as the code in Table 10.3 except that in step 3, the *n* schedules are the best *n* solutions chosen from the population after applying GA. The genetic algorithm starts with an initial population. It then performs the crossover operation and the population is updated. This is repeated a number of times.

10.6 Case Studies

The algorithms have been applied to the JSS problem and the TSP.

10.6.1 Job Shop Scheduling (JSS)

JSS involves scheduling of various operations relating to a number of jobs on a number of machines [2, 8, 4]. Different techniques exist in the literature for solving JSS [15]. Since the main focus of this chapter is parallel SA, we consider the SA algorithm for solving JSS. Each machine can process only one operation at a time. Each job consists of a sequence of operations in a pre-defined precedence order. The problem is to find a schedule having a minimum total time (cost), often called the 'makespan' of the schedule. An initial schedule is obtained for a given set of n jobs to be scheduled on m machines. The SA algorithm is applied to the initial schedule. The algorithm improves on the initial schedule by generating neighbourhood schedules and evaluating them. As the temperature is gradually reduced, the algorithm converges to a near optimal solution.

One of the major drawbacks of the SA algorithm is that it is very slow, especially when the search space is large. The algorithm's behaviour is also greatly influenced by the choice of the initial schedule. Attempts to parallelize the SA algorithm for JSS resulted in three different algorithms, namely the TMA, the LEA, and the MLEA [10, 12]. Although the TMA is problem-independent, it has not shown much improvement in speed-up. However, the quality of results produced is better. In the other two algorithms, the search space is divided by using a special technique called the 'locking edge technique'. These algorithms are highly problem-dependent and hence are applicable only to the JSS problem. It is for the above reasons that a problem-independent true distributed SA algorithm is attempted for development. One of the basic problems concerning distributed SA algorithms is that SA is inherently sequential. Hence, the option is to distribute the search space and let the algorithm run on a reduced search space on each node of the network. Since the division of the problem into sub-problems depends largely upon the characteristics of the problem, such an algorithm cannot be general in nature. Also, it is likely that one node gets into other nodes' search space, which is termed as collision. Such intrusions have to be treated separately. For these reasons, a static division of the search space among the nodes is not ideal.

10.6.1.1 Performance Study of the Algorithms

The performance of the CA and GCA are compared with that of sequential simulated annealing (SSA) for different sizes of the JSS problem (from 10 jobs on 10 machines to 20 jobs on 15 machines). The annealing algorithm used an initial temperature of 2000, and a temperature modifier of 0.95. Annealing was frozen when the best solution found did not change for the last 25 temperature changes or the temperature was reduced to zero. Table 10.6 shows the performance of SSA. The performance of the algorithms has been compared on the basis of two factors, namely, the execution time of the algorithm and the cost of the solution. It can be observed from Table 10.7 that CA performed very well in terms of both execution time and the quality of the solution as compared to the SSA algorithm. CA sometimes showed super-linear speed-ups. It can also be observed from Table 10.7 that at low problem sizes, GCA performed better than CA. This can be explained from the fact that at low problem sizes, GCA is able to give good initial solutions with a small overhead, whereas in the case of large problem sizes, the quality of the population is not appreciably improved by running GA for a fixed time period. In case of GCA, the time spent on GA can be increased to select good initial solutions. But the overhead of GA increases correspondingly. Hence, the optimal time to be spent on GA in the case of GCA can be found by conducting experiments by varying this time.

Table 10.6 Effect of Initial Solution on Simulated Annealing

Initial makespan	*Final makespan*	*Time taken(s)*
1076	896	498
1120	896	510
1284	905	610
1328	905	815

Table 10.7 Comparison of SSA, CA and GCA Performance (with Three Client Nodes)

Problem size	*SSA*		*CA*		*GCA*	
(jobs × m/cs)	*Time(s)*	*Cost*	*Time(s)*	*Cost*	*Time(s)*	*Cost*
10 × 10	577	968	307	931	199	971
10 × 15	1402	1603	803	1565	551	1683
10 × 17	2864	1548	128	1542	890	1538
20 × 10	7439	3309	3137	3309	3266	3434
20 × 15	6701	3391	2862	3391	1907	3391

Table 10.8 shows the relative performance of CA and GCA for a fixed problem size as the number of nodes is varied. As the number of nodes is increased, CA performed better than compared to GCA. This can be explained from the fact that GCA requires *n* good initial solutions generated by GA. As *n* increases, the quality of the initial solution decreases as GA is run only for a fixed initial time.

Table 10.8 Performance of CA and GSA for a Specific Problem Instance (Size = 10 × 15)

	CA		*GCA*	
No. of nodes	*Time(s)*	*Speed-up*	*Time(s)*	*Speed-up*
3	835	2.44	695	2.93
4	656	3.10	551	3.70
5	593	3.75	494	4.12
6	419	4.86	502	4.05
7	445	4.57	321	6.31
8	317	6.42	296	6.88
9	226	9.01	288	7.07

Table 10.7 shows the relative performance of CA and GCA as the problem size is increased keeping the number of nodes fixed. At low problem sizes, GCA performed better than CA. This can be explained by the fact that as the problem size increases, the quality of the initial solutions generated by GA by running it for a fixed amount of time is not good.

10.6.2 Traveling Salesman Problem (TSP)

The TSP is that of finding the minimum weight Hamiltonian cycle in a given undirected graph. The quality of solutions obtained for TSP by using SA is good, though obtaining them is time-consuming. It is in this context that parallel algorithms for SA for solving TSP are of practical interest. We have applied the GCA for the TSP and compared its performance with that of the sequential algorithm. The graph is represented as an adjacency matrix of inter-city distances. A valid initial solution for SA and the initial population for GA are obtained by a depth-first search.

The neighbouring solution for SA is generated by the method suggested in [6], wherein a section of the path chosen is traversed in the opposite direction. The genetic operator cross-over called edge re-combination [14] is applied to a pair of parent solutions to generate two offsprings and the best two out of the four solutions replace original ones. The required *n* initial solutions for the parallel SA are obtained by the genetic algorithm as explained earlier.

10.6.2.1 Performance Study of the Algorithms

Three sizes of the TSP, namely 50, 100, and 150, have been considered for the performance analysis of the GCA algorithm. The cooling schedule for SA employed an initial temperature of 1000 and a temperature modifier value of 0.99, and that for GCA, an initial temperature of 1 and a temperature modifier value of 0.99. GA was stopped when the solutions did not change over a fixed number of iterations. SA was also stopped when there was no improvement in the quality of the solution over a fixed number of iterations.

The relative performance between GCA and SA for TSP are given in Table 10.9. Three trials were taken on three problem instances and the average value of the cost and time are tabulated in each case. It is evident from Table 10.9 that GCA performed very well as the problem size increased and showed super-linear speed-ups at higher problem sizes. At lower problem sizes, the algorithm did not perform well, as the overhead of the genetic algorithm is high on a smaller search space. We also experimented with the dynamic switching from GA to SA. Instead of running the GA for a fixed amount of time depending upon the search space size, the improvement in the quality of the n best solutions in the population has been used as the criterion for switching from GA to SA. When the n best solutions in the population do not change for a fixed number of iterations, GA is stopped. These n initial solutions are taken and fed to the parallel SA algorithms on n nodes of the network. It has been observed that there is an optimum time up to which GA can be run before switching to SA. If GA is run for less than this optimum time, n good initial solutions are not found for SA. If it is run for more time, time is wasted in finding more than n good initial solutions. Thus switching based on the improvement of the n initial solutions can be seen to perform best (refer to Table 10.10).

Table 10.9 Comparison of GCA and SA Performance for the TSP

Problem size	*GCA*		*SA*		
(No. of cities)	*Time(s)*	*Cost*	*Time(s)*	*Cost*	*Nodes*
50	107	717	112	767	3
100	139	1136	308	1134	3
150	497	1336	1568	1218	3

Table 10.10 Comparison of Dynamic and Static Switching Performance for the TSP (with Three Nodes)

	Dynamic switching		*Static switching*	
Problem size	*Time(s)*	*Cost*	*Time(s)*	*Cost*
50	55	707	57	725
100	80	1286	95	1276
150	100	1802	128	1862

10.7 Conclusions

Two distributed algorithms for SA have been developed and have been applied to the JSS and TSP. The algorithms showed very good results as the problem space and the number of nodes employed increased.

References

1. Greening, Daniel R., "Parallel Simulated Annealing Techniques", *Physica D*, Vol. 42, pp. 293–306, 1990.
2. Van Laarhoven, P.J.M., E. H. L., Aarts, and Jan Karel Lenstra, "Job Shop Scheduling by Simulated Annealing", *Operation Research,* Vol. 40, pp. 113–125, 1992.
3. Janakiram, D. and K. Ganeshan, "DiPS: A System for Implementing and Analysing Distributed Algorithms", Tech. Rep. IITM-CSE-93-001, Indian Institute of Technology, Madras.
4. French, Simon, *Sequencing and Scheduling: An Introduction to the Mathematics of Job Shop,* Wiley, New York.
5. Eglese, R.W., "Simulated Annealing: A Tool for Operation Research" *Eur. J. Oper. Res.*, Vol. 46, pp. 271–281, 1990.
6. Allwright, James R.A. and D.B. Carpenter, "A Distributed Implementation of Simulated Annealing for the Travelling Salesman Problem", *Parallel Comput.*, Vol. 10, pp. 335–338, 1989.
7. Decker, Keith S., "Distributed Problem Solving Techniques: A Survey", *IEEE Trans. Systems Man Cybernet*, Vol. 17, pp. 729–739, 1987.
8. Adams, J., E. Balas, and D. Zawack, "The Shifting Bottleneck Procedure for Job Shop Scheduling", *Mgmt. Sci.*, Vol. 34, pp. 391–401, 1988.
9. Abramson, David, "A Very High-speed Architecture for Simulated Annealing", *IEEE Comput*, pp. 27–36, May 1992.
10. Ganeshan, K., "Designing and Implementing Flexible Distributed Problem Solving Systems", M.S. Thesis, Department

of Computer Science and Engineering, Indian Institute of Technology, Madras, 1993.

11. Syswerda, Gilbert, "Schedule Optimization Using Genetic Algorithms", L. Davis (ed.), *Handbook of Genetic Algorithms*, pp. 332–349, 1991.

12. Krishan, K. Ganeshan, and D. Janakiram Ram, "Distributed Simulated Annealing Algorithms for Job Shop Scheduling", *IEEE Trans. Systems Man Cybernet*, Vol. 25, No. 7, pp. 1102–1109, July 1995.

13. Gu and X. Huang, "Efficient Local Search with Search Space Smoothing", *IEEE Trans. Systems Man Cybernet*, Vol. 24, No. 5, pp. 728–735, May 1994.

14. Whitley, Darrel, Timothy, Starkweather and Daniel, Shaner, "Schedule Optimization Using Genetic Algorithms", Lawrence Davis, (ed.), pp. 351–357.

15. Nowicki, E. and C. Smutnicki, "A Fast Tabu Search Algorithm for the Job Shop Problem", Report 8/93, Institute of Engineering Cybernetics, Technical University of Wroclaw, 1993 (To appear in *ORSA J Comput.*).

Chapter 11

Epilogue DOS Grid: Vision of Mobile Grids*

11.1 Introduction

Mobile computing brings about a new paradigm of distributed computing in which communication may be achieved through wireless networks and users can compute seamlessly even as they move from one environment to another. It is apparent that mobility affects the computational, data and transactional model, and the communication paradigm of the distributed mobile systems. The impact of the huge growth of the resource-constrained device goes beyond networking issues such as bandwidth and connectivity, and directly effects computing, data, and service management. The recent technological advancements in computing, wireless communications, networking and electronics have embedded processing power, storage space and communication capabilities in electronic devices of day-to-day use, leading to the era of ubiquitous computing. This trend has led to the pervasiveness, invisibility and mobility of computing nodes [1]. These factors have made the researchers look at mobile devices as both providers of services and consumers of services.

The advancement in technology has enabled mobile devices to become information and service providers by complementing or

*D. Janakiram, M.A. Maluk Mohamed, A. Vijay Srinivas, P. Kovendhan, M. Venkateswara Reddy replacing static hosts. Such mobile resources are highly essential for on-field applications that require advanced collaboration and computing. This creates the need for the merging of mobile and grid technologies, leading to a mobile grid paradigm. The key idea in building the mobile grid is to integrate the computational, data and service grids. Thus, a mobile device from anywhere and any time can utilize large computing power, required resources and services seamlessly. Simultaneously, the device could also be providing location-sensitive data to the grid. This has made us look at building the middleware for a mobile grid that transparently manages and bridges the requirement of the mobile users and the actual providers.

We are currently working towards our vision of integrating data grids with the mobile grid, thus seamlessly integrating data, computing and service grids. In addition to the data from regular sources in the data grid, we are looking at integrating data from wireless sensors into the mobile grid. Such an enhanced mobile grid will be useful for a large class of applications. Consider a scenario wherein inspection field engineers collect bridge maintenance data through various modes like sensors and visual data. The data may be huge so that it would have to be stored in data repositories. Further, the data collected by each engineer has to be compared and correlated with the data collected by other engineers as well as with the data collected from previous inspections. Thus, non-trivial computation is required to decide whether further data needs to be collected.

A key feature of the mobile grid is the monitoring system. The mobile grid monitoring system is used to maintain information about resources. It also enables task scheduling and data management, and handles various kinds of failures. To the best of our knowledge, there is no monitoring system for mobile grids. Further, existing grid monitoring systems have difficulty in scaling due to centralized components. We use a peer-to-peer architecture that avoids such centralized components. The placement of replicas that corresponds to file instances in the data grid is also handled by the mobile grid monitoring system. We enable files to be read-write (RW), unlike existing data grids which assume read-only (RO) files. In order to handle the consistency of replicas (and to provide other middleware-related services), we have built the mobile grid system over a wide-area shared object space called Virat [2].

The rest of the chapter is organized as follows: Section 11.2 explains the overview of the mobile grid and discusses how scalability and consistency are ensured in the proposed paradigm. Section 11.3 describes the Mobile Grid Monitoring system in detail, including the components of the monitoring system. Section 11.4 gives a sample health care application scenario. Section 11.5 compares the proposed model with the already existing similar work. Section 11.6 concludes the chapter.

11.2 DOS Grid

11.2.1 Overview of the Mobile Grid

The mobile grid is visualized as a cluster of clusters. In the proposed model, the nodes are encapsulated as objects called 'Surrogate Object' (SO). The SO encapsulates all the characteristics and properties of a device fully, such as the computing power, memory availability, bandwidth, and all other resources and services associated with the device. The representation of the characteristics of the devices in the SO is made as attributes, methods and sub-objects. The attributes of the SO include the computing capability of the node, the memory capability and the bandwidth of the medium by which the node is connected. The methods and the sub-objects of the SO represent the services and other resources that are offered by the node. In addition, each SO encapsulates the security policy and agreement for each of the services that is associated with that node, which will specify how and by whom the service may be used. By encapsulating the participating nodes, in distributed objects, the grid is transformed from a collection of nodes, offering and consuming services, into Distributed Shared Object (DSO) space.

Each cluster is coordinated with a designated node acting as a Cluster Head (CH). The CH maintains all the repositories related to the trading and naming services of the DSO, and handles the service discovery [3]. The CHs coordinate among the other neighbouring CHs in a peer-to-peer fashion. The mobile grid as DSO space is shown in Fig. 11.1. In order to have a unified design, the static nodes are also represented as objects in the DSO. As the MSS are already loaded with maintaining the information related to the MH which are within its cell, the neighbouring static node is designated as the CH of the cluster. In some cases the MSS and CH may be the same.

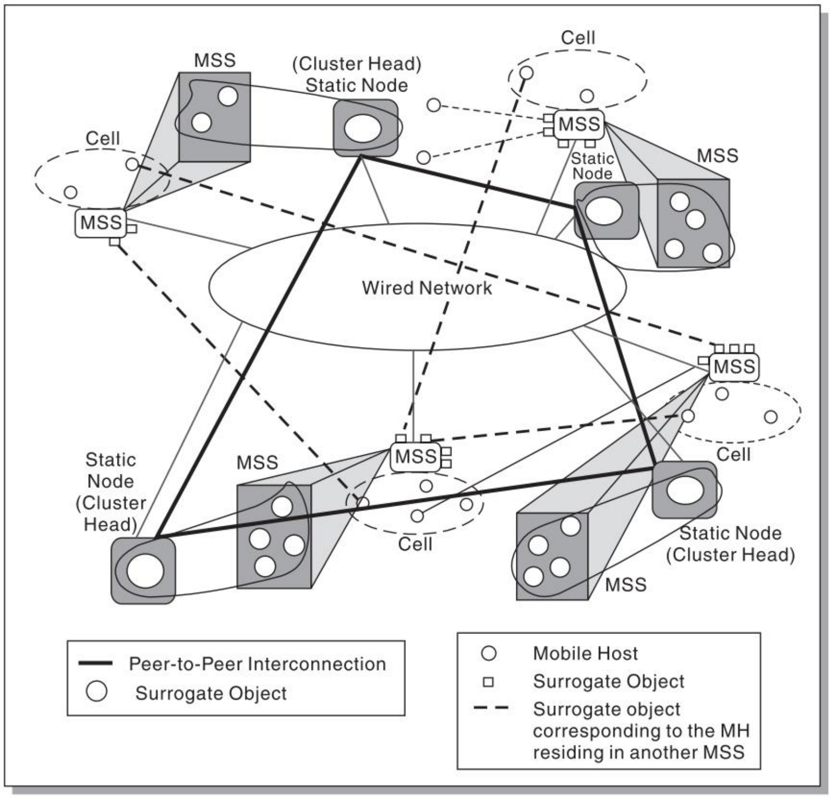

Cell
MSS
(Cluster Head)
Static Node
Cell
MSS
Static Node
MSS
MSS
Wired Network
MSS
Static Node (Cluster Head)
MSS
MSS
Cell
Cell
Static Node (Cluster Head)
MSS
Peer-to-Peer Interconnection
Surrogate Object
Mobile Host
Surrogate Object
Surrogate object corresponding to the MH residing in another MSS

FIG. 11.1

Mobile Grid Architecture with Surrogate Objects

The proposed SO model of the grid enhances the availability and helps in providing the current location of the mobile devices. When the mobile device acts as an information service provider, depending upon the nature of the service requested, the monitoring system of the mobile grid decides whether to contact the SO of the corresponding device or the device itself. Contacting the device directly will lead to consumption of the constrained resources like battery and bandwidth. In the case of the services offered by the SO, it would suffice if the corresponding SO is contacted to get the information. In addition, the SO can be replicated to prevent congestion in the network and to improve scalability of the system. The major advantage of the paradigm is that the network connectivity need not be continuous because connections are required only to inject SO from mobile nodes into the wired network. With the SO being fully autonomous, users can access services even if the node disconnects because the SO delivers the results upon re-connection. The proposed model virtualizes all the resources and services offered by the participating nodes as services.

The proposed approach significantly helps in realizing a distributed and decentralized infrastructure of SOs that work on behalf of the participating devices and are hosted by the wired network. With the SO, the MH movements do not affect service provisioning as the entire state of the device is stored and maintained. The model helps in achieving the properties of dynamicity, asynchronicity, autonomy and security.

Some memory-constrained devices such as mobile phones could also be participating in the mobile grid. In such a case, the user of the mobile device should be able to store the data in some other databases located elsewhere in addition to the data stored locally. Thus, the proposed mobile grid can also efficiently handle storage and data access from federated databases. The meta-data contains information about file instances, the contents of file instances, and the various storage systems contained in the data grid. These meta-data usually refer to application meta-data. These meta-data are wrapped with wrappers and made as objects in the DSO. The meta-data objects are registered by using trading services.

11.2.2 Scalability and Consistency Issues

Middleware services such as naming and trading as well as replica object management in the grid are handled through a wide-area shared object space that we have built named as Virat [2]. Virat uses an independent checkpointing and lazy reconstruction mechanism to handle failures of object repositories. The object repositories (one per cluster) are responsible for the cluster level management of replicas. Communication between the object repositories themselves is through a peer-to-peer protocol. This is useful for locating objects or services across clusters. Virat also uses a data-centric concurrency control mechanism to realize various consistency schemes such as serializability and causal consistency. Virat has been extended to a shared event space wherein events can be created, published and subscribed. Events can be delivered in causal or serializable orderings on the basis of application requirements.

Scalability is a key issue in distributed systems, especially in mobile grids as the number of devices can be quite high. One of our key observations is that scalability, consistency and availability need to addressed together, not in isolation from each other. It is well-known that in the presence of network partitions, both consistency and availability cannot be completely attained in purely asynchronous systems [4]. Availability has been quantified in [5] and its trade-off with consistency has been studied. However, both [4] and [5] do not address the scalability issue.

One dimension of consistency is the δ value, the number of updates that can be buffered by a replica before updating other replicas. This has been related to availability in [6]. However, another dimension of consistency, namely update ordering, has not been considered. Various consistency criteria can be realized on the basis of update ordering. These include serializability, causal consistency and Pipelined Random Access Memory (PRAM) consistency. The idea is that given these two dimensions of consistency, there is a trade-off between scalability and availability. We have come up with an upper bound on scalability (in terms of productivity) for a given availability and the two dimensions of consistency, for specific workload and faultload combination. This theoretical upper bound is difficult, if not impossible, to achieve in practice. We are currently conducting performance measurements to evaluate the practical scalability of Virat. We are also optimizing Virat to make the scalability closer to the theoretical upper bound.

11.3 Mobile Grid Monitoring System

Monitoring is the act of collecting information concerning the characteristics and status of resources of interest. Grid resources may dynamically join and leave, resulting in varying membership over time. Even in fairly static conditions, resource availability is subject to failures. Due to such a transient nature of the grid, the system must support the finding and keeping track of the required resources dynamically. This is the main purpose of Grid Information Services (GIS). This requires a process called monitoring, which systematically collects the information regarding the current and past status of the grid resources to satisfy the users' need.

Several groups are developing grid monitoring systems [7]. In most of these monitoring systems, the monitoring system is a part of the discovery system. However, in our proposed approach, the publishing, discovery and handling of resources are done by the DSO structure which is considered as a platform to build the mobile grid infrastructure. The monitoring system resides over the shared space in the peer-to-peer layer.

In order to incorporate intelligence into the mobile grid, we require a monitoring system that manages the vast heterogeneous resources including those offered by the mobile devices across administrative domains. The mobile grid monitoring system essentially helps in scheduling and task allocation for parallel computing, in enforcing Quality of Service (QoS) and Service Level Agreements (SLA), in identifying the cause of performance problems, optimized resource usage and fault detection, in addition to building prediction models of mobile device movement.

Different kinds of data are collected from the different components that make up the mobile grid.

The monitoring information traffic can be huge and with only one monitoring manager, it becomes a bottleneck for the entire system. Hence, the hierarchical monitoring structure was considered by most of the existing monitoring systems. However it is proved in [8] that the scalability of peer-to-peer structure is better than a hierarchical structure. Hence the CHs associated with each cluster, which forms the peer-to-peer overlay, share the monitoring activities among themselves.

The proposed Mobile Grid Monitoring System provides the following features:

1. *Mobile Host (MH) monitoring:* Data regarding the location of the device, connectivity, mobility, signal strength, etc. at different times of the day can be monitored and used for predicting the future values of these parameters. This could help in characterizing the movement pattern of the devices. This information can be used in scheduling to provide guaranteed QoS and SLA. In addition, information regarding load, battery life, available memory, etc. is also collected. When these parameters cross the threshold value, the MH sends the associated information asynchronously to the monitoring system. The mobile node parameters are also sent on the basis of the request from the monitoring system. Mobile device monitoring can be done by monitoring the SO associated with the mobile device.

2. *Providing dynamic space allocation and file management on shared storage components on the grid:* They provide storage reservation and dynamic information on storage availability for data movement, and for the planning and execution of grid jobs. This is designed to facilitate the effective sharing of files, by monitoring the activity of shared files, and making dynamic decisions on which files to replace when space is needed.

3. *Data store monitoring:* The access patterns of the data can be observed to assist in replica management. Data regarding data movement overhead and data lifetime can also be

collected. This can be used to decide whether the task must be scheduled near the data store or data moved near the task. It is also possible to maintain the meta-data of the different data stores and their elements in order to assist in storage management, handling requests, etc. The meta-data could also contain information about the locks on the different data elements.

4. *Static Host (SH) monitoring:* We collect the CPU load, available memory, bandwidth details, etc. from the host. The hosts can either be monitored continuously or on being triggered by detecting network activity on a particular port. This information can be used by the scheduler to choose the donor.
5. *Process monitoring:* Changes in the process state can be monitored.
6. *Application monitoring:* Checkpoints can be inserted into the application to capture the intermediate state of the application and the data required for performance analysis.

Any query to a mobile device is routed through its associated SO. The object reference of the SO is obtained from the naming service of the DSO by querying for the IP of device as name [3]. The query is forwarded to the MH, with the help of the current location information available at the SO. In case the MH is not reachable due to reasons like out-of-coverage, the query is handled at the SO level. The monitoring system stores the historical data related to the MH movement pattern in the repository for characterizing the movement pattern. This helps in scheduling tasks and storing data.

In our proposed model, the monitoring system will observe the requests and try to find out from which place and for which file the requests have come. Based on the observations, the lifetime of the file, data transfer time and the available space in the data store, the system chooses the optimal location. This is done by maintaining the history in the monitoring system about the requested queries and the lifetime of the data which they are accessing to predict the location as well as to initiate the replica dynamically. Although the probability of prediction is high, still even if it fails, it affects only the performance of the system, not its correctness.

The monitoring daemon, which collects the monitoring data, runs on the static nodes of the mobile grid. Mobile nodes run the exporter daemon, which exports the monitoring data of the mobile nodes to the mobile support station. The monitoring system resides on the CH of the cluster and maintains a monitoring data repository to store the data observed within the cluster. As the CHs interact among themselves over a peer-to-peer overlay, the monitoring system is scalable, fault-tolerant and ensures automatic reconfiguration.

11.4 Health Care Application Scenario

One of the possible scenarios wherein we can envision the integration of sensors, mobile nodes and data grids is the following health monitoring and treatment example. Patients could have sensors embedded inside their bodies to keep checking for specific data such as blood pressure, cholesterol level, etc. When local sensor data exceeds certain pre-defined thresholds, the sensor passes the data to a nearby mobile device, possibly the patient's hand-held device. The device is part of our mobile grid and can utilize and provide services. It uses the grid to aggregate data from multiple sensors from the same patient, uses historical information and some computation to decide if this pattern (combination of data values from various sensors on a patient) is abnormal and requires emergency handling. If this is the case, the details of the patient are collected from the grid and forwarded to a health care centre. Based on the location of the mobile device which sent the data to the grid, the location of the patient is tracked.

A computer system at the health care centre (it is also part of the grid) locates a nearby ambulance by querying the grid. The ambulance carries some mobile devices and can be directed to the patient's location as soon as possible. This computer system also sends the patient's details to the closest available physician. The physician could then pull out the medical history of the patient from the data grid (which contains historical data of all patients treated at various medical centres) and, based on the history and the current problem, come up with a strategy for immediate treatment. This treatment strategy could then be passed on to the paramedics who are on the ambulance approaching the patient. This could help the paramedics to be totally prepared and to start treating the patient immediately on reaching him. Further, the mobile grid would also help in assembling the required medicines and injections to be administered to the patient when they arrive on the scene.

11.5 Related Work

Our work is unique in that no major mobile grid systems exist, to the best of our knowledge. Consequently, existing grid systems do not address the issue of mobile devices integration. Further, grid monitoring systems do not address what to monitor and how to monitor in mobile environments. Thirdly, our idea of integrating data grids as well as data from sensors into the mobile grid is a further distinguishing feature of our work.

Very few works (such as [9,10,11]) are done in mobile grids. But none of these chapters addresses the issues related to mobility, which leads to an availability problem. They had also specifically looked only at the computing nature of the mobile grid. [9] discusses the challenges that arise due to the integration of mobile devices into a grid. [10] provides a proxy-based approach for the sharing of computing power among the participating mobile devices. The model considers only the mobile *ad hoc* networks. [11] comes out with an architecture using Mobile Agent (MA) technology, wherein the mobile agents are used as a communication primitive. The chapter focuses on mobile devices using the grid and not as members of the grid.

Some of the major projects involved in the integration of computational and data grids worldwide are CERN's Openlab

[12], NASA's Information Power Grid [13], European Union's DataGrid [14], Particle Physics Data Grid (PPDG) Collaboratory Pilot [15] and Queens University's DataCentric Grid project [16].

The CERN Openlab is a collaborative project between CERN, the European Organization for Nuclear Research and industrial partners, and aims to develop data-intensive grid technology to be used by several scientists worldwide working at the next-generation Large Hadron Collider. The Large Hadron Collider (LHC), being constructed by CERN, will be the most powerful particle accelerator ever built. It will commence operations in 2007 and will run for up to two decades. Detectors placed around the 27 km LHC tunnel will produce about 15 Petabytes of data per year. These huge amounts of data must be stored in a distributed fashion and made accessible to thousands of scientists. This requires a lot of resources. CERN has therefore launched the LHC Computing Grid (LCG), whose mission is to integrate tens of thousands of computers at dozens of participating locations worldwide into a global computing resource.

DataGrid, a project funded by European Union, is working towards building the next generation computing infrastructure providing intensive computation and analysis of shared large-scale databases, from hundreds of TeraBytes to PetaBytes, across widely distributed scientific communities. The grid middleware provided by the Globus Toolkit is enhanced to work better with large data sets, many files and many users distributed over several sites and organizations. Some of the research areas being addressed by this project are work scheduling, data/replica management, monitoring services, fabric management, storage management, integration testbed and support.

NASA's Information Power Grid (IPG) is a high-performance computation and data grid that integrates geographically distributed computers, databases and instruments. On top of the basic grid services, the IPG middleware shall build both generic and discipline-specific workflow management systems that will carry out the human defined protocols, such as multi-disciplinary simulations and data analysis, and global data cataloguing and replica management systems needed to manage the data for these scenarios.

The Datacentric Grid Project aims to design and implement grids for data-intensive operations in which data is moved as little as possible. The volume of data quadruples every 18 months, while the available performance per processor doubles in the same time period. Thus, moving data to the program would not scale in the long run. It is much more effective to divide programs into separate pieces and send them to the data. This requires a data-centric view of computation, rather than the conventional processor-centric view. This would require the data repositories to be fronted by large computer servers to process their data. The data-centric grid would also require a new programming model, possibly embodied in a kind of query language.

The Particle Physics Data Grid Collaboratory Pilot (PPDG) is developing and deploying production grid systems with effective end-to-end capabilities by integrating experiment-specific applications, grid computation technologies and storage resources. Sustained production data movement over the grid has resulted in better data transfer throughput, reduced operational effort, and a paradigm shift for distributed data processing from manual to automated bulk file transfers.

In Data Grids [17], normally the scheduler schedules the tasks on to computational elements which are in the near proximity of the storage elements. If no computational element is available, then they try to replicate the data store towards the computational task. They do not optimize replica location.

11.6 Conclusions

We are envisioning a mobile grid in which any device (including wireless sensors or other mobile devices) can seamlessly provide or access data, computation and other services anywhere anytime. This is enabled by integrating data grids and sensor data into our existing mobile grid. We have also come up with a mobile grid monitoring system that enables the grid to optimize resource utilization and scale, in addition to handling mobile device constraints.

We have currently simulated the mobile grid. We are currently building the hardware interfaces and the software required to realize the grand vision. An interesting direction for future research is to focus on specific aspects of the mobile grid such as scheduling and to come up with efficient algorithms for the same. We are also looking at providing the right abstraction for programming the mobile grid.

References

1. Satyanarayanan, M., "Pervasive computing: Vision and challenges", *IEEE Personal Communications*, pp. 10–17, August 2001.
2. Srinivas, A. Vijay, D. Janakiram and Raghevendra Koti, "Virat: An Internet Scale Distributed Shared Memory System", Technical Report IITM-CSE-DOS-2004-03, Distributed and Object Systems Lab, Indian Institute of Technology, Madras, January 2004.
3. Janakiram, D., M.A. Maluk Mohammed, A. Vijay Srinivas and Mohit Chakraborty, "SOM: A Paradigm for Location and Availability Management in Distributed Mobile Systems", Technical Report IITM-CSE-DOS-2005-01, Distributed and Object Systems Lab, Indian Institute of Technology, Madras, January 2005.
4. Gibert, Seth and Nancy Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services", *ACM SIGACT News*, Vol. 33, No. 2, pp. 51–59, June 2002.
5. Yu, Haifeng and Amin Vahdat, "The Costs and Limits of Availability of Replicated Services", Proceedings of the ACM Symposium on Operating System Principles (SOSP), Banff, Canada, October 2001.
6. Zhang, Chi and Zheng Zhang, "Trading Replication Consistency for Performance and Availability: An Adaptive

Approach", Proceedings of the 23rd International Conference on Distributed Computing Systems, Providence, Rhode Island, USA, May 2003.

7. Zanikolas, Serafeim and Rizos Sakellariou, "A Taxonomy of Grid Monitoring Systems", *Elsevier Journal on Future Generation Computer Systems*, Vol. 21, pp. 163–188, 2005.

8. Foster, Ian and Adriana Iamnitchi, "On Death, Taxes, and the Convergence of Peer-to-peer and Grid Computing", Proceedings of the Second International Workshop on Peer-to-peer Systems (IPTPS 2003), Springer-Verlag, February 2003.

9. Phan, Thomas, Lloyd Huang, and Chris Dulan, "Challenge: Integrating Mobile Wireless Devices into the Computational Grid", Proceedings of the Eighth Annual International Conference on Mobile Computing and Networking, ACM SIGMOBILE, September 2002.

10. Hwang, Junseok and Praveen Aravamudham, "Middleware Services for P2P Computing in Wireless Grid Network", *IEEE Internet Computing*, pp. 40–46, July-August 2004.

11. Bruneo Dario, Marco Scarpa, Angelo Zaia and Antonio Puliafito, "Communication Paradigms for Mobile Grid Users", Proceedings of the Third IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03). IEEE Computer Society, 2003.

12. Grey, Francois "The CERN Openlab: A Novel Testbed for the Grid", *CERN Courier*, Vol. 43, No. 8, pp. 51–59, September 2003.

13. Johnston, W.E., D. Gannon and B. Nitzberg, "Grids as Production Computing Environments: The Engineering Aspects of NASA's Information Power Grid", Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing, Redondo Beach, California, August 1999.

14. Segal, B. "Grid Computing: The European Data Grid Project", Proceedings of the IEEE Nuclear Science Symposium and Medical Imaging Conference. Lyon, France, October 2000.

15. Skow, Daniel, "Particle Physics Data Grid Collaborative Science (White Paper)", *http://www.ppdg.net/docs/SciDAC/20050118_PPDG_WP.pdf.*

16. Skillicorn, D.B., "The Case for Data-centric Grids", Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS 02); Fort Lauderdale, Florida, April 2002.

17. Chervenak, A., I. Foster, C. Kesselman, C. Salisbury and S. Tuecke, "The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Data sets", *Journal of Network and Computer Applications*, Vol. 23, pp. 187–200, 2001.