A JAVA BASED LOAD BALANCING FRAMEWORK FOR NETWORK PARALLEL APPLICATIONS

BY

IRFAN AHMED ILYAS

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

APRIL 2000

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **IRFAN AHMED ILYAS** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Deam of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE.**

THESIS COMMITTEE

Dr. Muslim Bozyigit (Thesis Advisor)

Dr. Jarallah S-AlGhamdi (Member)

Dr. Onur Toker (Member)

Department Chairman

Dean, College of Graduate Studies

27/5/2002

Date

Dedicated to

My Parents, Teachers, Friends

and

all those who contributed to my achievements

ACKNOWLEDGEMENTS

In the name of Allah, Most Gracious, Most Merciful

All Praise is due o ALLAH to whom belongs the dominion of the Heavens and the Earth. Peace and mercy be upon His Prophet who gave us the way of peace and real success in our lives. My due thanks are for ALL MIGHTY who furnish me with the essential knowledge and patience to carry out such a challenging work.

At the beginning, I must acknowledge the continuos motivations and prayers of my parents which are always there for helping me reaching at major milestones of my life. I would also acknowledge the love and encouragement of my sisters and brother which help me immensely in pursuing my studies here at KFUPM, far away from them.

I would like to offer my indebtedness and sincere appreciation to my thesis advisor Dr. Muslim Bozyigit and must say that no word of thanks would be sufficient for his guidance. During the course of the research, he was always available and ready to help even at very odd hours (late in the evening). Special thanks are due to my thesis committee members Dr. Jaralla Al-Ghamdi and Dr. Onur Toker for all their cooperation and advice.

Thanks are due to my friends Shafaat, Khursheed, Raza, Arshad, Saaqib, Shahid and all others with whom I spent some memorable moments.

Finally, acknowledgement is due to King Fahd University of Petroleum and Minerals for the support and resources provided in this research.

Irfan Ahmed Ilyas

Contents

List of Tables

List of Figures

List of Algorithms

THESIS ABSTRACT

Name: Irfan Ahmed Ilyas
Title: A JAVA BASED LOAD BALANCING FRAMEWORK FOR NETWORK PARALLEL APPLICATIONS
Degree: MASTER OF SCIENCE
Major: Computer Science
Date of Degree: April 2000

Network of workstation (NOW) environments provide attractive scalability in terms of computation power and memory size. However in their early years, they could not attract much of the attentions as competitors against existing parallel machine environments. With the rapid advances in new high-speed computer network technologies, a NOW is becoming increasingly competitive. The main reason is the much smaller cost/performance ratio and the high availability of these environments. However, because of the additional issues involved due to their loosely coupled nature, NOWs are harder to program for parallel applications. Some of the issues to be considered are degree of heterogeneity in architecture and operating systems, uneven external load, communication overheads and high variation in the system performance.

Load balancing problem in parallel and distributed computing domain addresses the assignment of parallel tasks among available interconnected hosts in proportion to their performance and existing loads on them. This assignment can be static- done at compile time, or dynamic- done at run-time. While static balancing avoids the run-time scheduling overhead, in a NOW environment a dynamic scheduling approach is needed. Despite the overheads involved, dynamic load balancing techniques are proved to improve the performance of the system to a substantial degree. Libraries such as PVM and MPI have facilitated development of load balanced parallel applications over heterogeneous platforms.

Java, a web based language, allows the execution of platform neutral code anywhere in a distributed setup. Its supports for networking, multi-threading, object serialization and mobile code (using dynamic class loading feature) open new horizons in distributed application development.

This research is aimed to test Java as a choice for parallel and distributed load balancing systems. A class framework, named JLBS (Java Load Balancing System), has been developed for supporting load balanced distributed applications.

The JLBS system has been tested using a number of hypothetical applications and one real application (matrix multiplication). Our experimental results proved the system to be scalable and having decent level of speedups (in execution times) for parallel applications.

Keywords: Network of workstations, dynamic load balancing, parallel processing, JVM

King Fahd University of Petroleum and Minerals, Dhahran

April 2000

خلاصة الرسالة

أسم الطالب : عرفان أحمد الياس

عنوان الرسالة : نظام توزيع الحمل للتطبيقات المتوازية على الشبكة باستخدام لغة جافا

التخصـــص : الحاسب الآلي

تاريخ الشهادة : أبريل ٢٠٠٠م

الشبكات الحاسوبية تتميز بمرونتها في الذاكرة والأداء ، ولكنها لم تكن تحاكي وتقارن قـــدرة الأجـــهزة المتوازية السنوات الماضية . ولكن مع التطور الحالي الهائل للشبكات أصبحت تنافسها لتميزها في قلـــة الســـعر وتوفر الشبكات . ولطبيعة الشبكات تبقى البرمجة المتوازية فيها أصعب . ومن هذه الأشياء التي يجب مراعاتها : التغاير في الشكل (أو البناء) وفي نظام التشغيل وفي الأداء وفي الأعمال الخارجية وكذلك قوة وقدرة الاتصال .

المشكلة هي في القدرة على توزيع الأعباء (أو الأعمال أو المسئوليات) بين الأجزاء (أو الأجهزة) المختلفـــة بشكل متوازن وذلك إما بتوزيعها مرة واحدة أو ما يمكن تسميته بتوزيع مستمر ونشط أثناء القيام بالعمل، وهذا الأخير هو المطلوب في الشبكات لأنه أثبت تحسن عالي في الأداء برغم كل الصعوبات . وقد ســـهلت بعـــض المرجعيات مثل **PVM** و **MPI** عملية التوازن في توزيع المهام على الأوساط والأجهزة المتغايرة .

والقدرات الكامنة في لغة جافا مثل القدرة على التعامل مع الشبكات ومختلف الأجهزة والقـــدرة علـــى التعامل مع الأشكال والقدرات الحركية هي التي فتحت آفاق جديدة في عالم الشبكات والبرمجيات وتطويرها.

وهذا البحث هو لاختبار قدرة جافا على حل مشكلة التوازن في توزيع المهام على مختلف الأجهزة ، وقـد صمم لذلك هيكلة نظامي رمز له **JLBS** . وقد اختبرناه بواسطة عدة برمجيات . وقد أثبتت نتائج الاختبـــارات تحسن في سرعة الأداء والمرونة .

هذا ونسأل الله العلي القدير أن يبارك في هذا العمل ويجعله وسبله إلى الخير والصلاح .

المصطلحات : شبكة محطات العمل ، موازنة الأحمال، تطبيقات متوازية ، الصمود أمام الأعطال ، ارتجال التطبيقات .

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران المملكة العربية السعودية

أبريل ٢٠٠٠م

Chapter 1

Introduction

In the world we are living, computing industry is proved to have the most rapidly changing trends. With its start in the seventeenth century, when a young French mathematician, Blaise Pascal, constructed the first widely known mechanical adding machine, till now with the existence of the world wide computing village, the Internet, the pace of changing trends is nearly impractical to be tracked down. However, the world computing requirements are constantly growing resulting in grand challenge problems requiring massive amounts of computations.

Parallel processing deals with the solution of grand challenges problems. Some of the problems that requires massive computations are related to:

- Numerical Weather Prediction
- Oceanography
- Socio-Economic and Government applications
- Engineering Applications, such as
    - Finite Element Analysis
    - Computational Aerodynamics
    - Plasma Physics
- Artificial Intelligence Applications, such as
    - Image Speech and Pattern processing
    - Computer Vision
    - Robotics

For solving such problems, the trend was to use special purpose dedicated architectures, which consist of a number of high performance tightly coupled processing elements. Such solutions are naturally costly and of limited utility as designed for special purpose use

(targeting some specific area). Thus frustrating levels of performance/cost ratio was resulted for such solutions.

The existence of *distributed systems* gave new horizons to the area of parallel computing. These systems consist of a number of computing architectures (single or multi processor based workstations) interconnected together by means of wire/wireless links. Their widespread diffusion motivates the attempts to exploit the intrinsic parallelism in these computing environments, for solving the existing grand computation challenges. The idea is further encouraged by the advances in processor technology (like the development of RISC, Reduced Instruction Set Computers, architecture) as well as high speed network technologies (e.g. ATM, Asynchronous Transfer Mode). Just to have a practical estimate of the advances in the processors, their processing power has a 500-fold increase in the last thirteen years whereas for communication domain the improvement is estimated to be five orders of magnitude for the same period [18].

Moreover, studies for widely available distributed environments (LAN) have shown that for a large percentage of their lifetimes, the machines in such environments are used for small user tasks like reading e-mails, editing files etc. Thus these systems are proved to demonstrate an average idle time of at least 90% even during peak hours of their use [8]. The use of such system for solving grand computation problems also helps utilizing existing under utilized computing.

1.1 Parallel Architectures

By definition, a parallel program contains more than one process (task or thread), which can run in parallel to solve a problem. Clearly, their parallel run needs the availability of multiple processing elements or processors, connected with each other in some fashion, in order to have some coordination among the running processes.

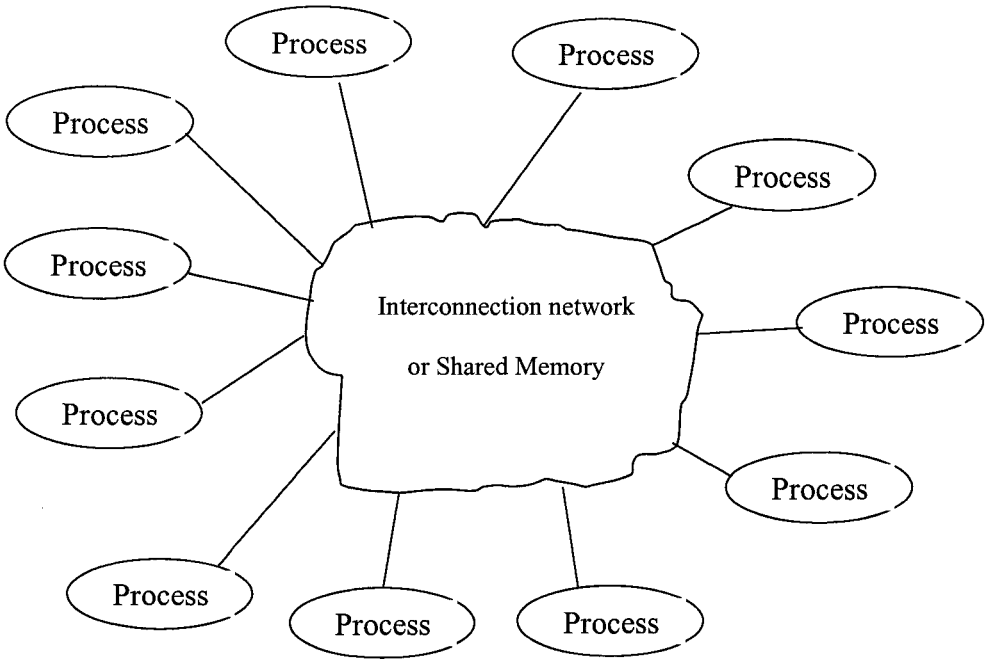


Figure 1.1 General model of a parallel computer

A parallel computer provides a collection of two or more processors connected to one another through some interconnecting media. Figure 1.1 shows the general form of such an interconnection. The manner in which the processor operates then further classifies the architecture as MIMD (Multiple Instruction Multiple Data) or SIMD (Single Instruction Multiple Data) System [29].

1.1.2 MIMD Parallel Systems

In MIMD Systems, the processors operate independently of one another but with occasional pauses to synchronize their processes (required for coordination logic). The processors can have individual (distributed) or a shared memory space available. Thus

processes running on a MIMD system can coordinate using messages (in distributed memory machines) or using shared memory space.

Examples of MIMD machines are Intel iPSC series, nCUBE series, Sequent Symmetry series, IBM SP2 and Cray T3D. A network of workstations, connected through high-speed links, could also be considered as a MIMD architecture.

1.1.3 SIMD Parallel Systems

In SIMD Systems, each processor runs same processes but on a different data set. They are tightly synchronized as they either run the same program or become idle. Examples of SIMD machines are the Thinking Machines Inc. Connection Machine (CM) series and the Maspar Computer Corp. MP series.

In SIMD systems, all processors must have individual memories with no concept of a central (shared) memory. During a program execution, the subdivision of program data for every processor is copied into its attached memory. Thus SIMD systems essentially require message passing mechanism to have process coordination.

1.2 Parallel Programming Paradigms

A parallel programming paradigm dictates the way parallelism is controlled in a particular parallel program. Generally speaking, a parallel program can follow one of two general styles to properly synchronize its parallel parts: control flow style or data-parallel style [29].

1.2.1 Control Flow Style

A control-flow program supports more than one thread of control at the same time. It means a single program can perform different operations (in different processes) in the

same time interval. The execution of such programs is mainly governed by the program control rather than by the availability of data. Such programs can be covered under MIMD class of parallel programs.

1.2.2 Data Parallel Style

A data parallel program performs identical operations on different data sets, obtained by partitioning the complete application's data. However the program execution may be tightly coupled, with tasks synchronized after running a certain step, or it may be slightly relaxed to permit entire procedures to complete in between synchronizations. The former approach corresponds to a SIMD (Single Instruction Multiple Data) program and the later to a SPMD (Single Program Multiple Data) program.

1.2.3 Message Passing Vs Locking

The discussion of paradigm can be further narrowed by considering the method adopted for process synchronization. Two approaches are possible: *using message passing* or *using lock mechanisms with shared memory*. A control flow program can implement any one of the these while a data-parallel program essentially needs a message passing approach (for distributing data among the processes).

Figure 1.2 illustrates possible programming paradigms that can be adopted in parallel program architectures. The following three could be realized:

- control-flow with message-passing,
- control-flow with locking, and
- data-parallel with message-passing.

The selection of a paradigm is dependant on the architecture of the underlying hardware. MIMD architectures are the most general parallel machines, because they can implement all possible paradigms with some inefficiency due to overhead.

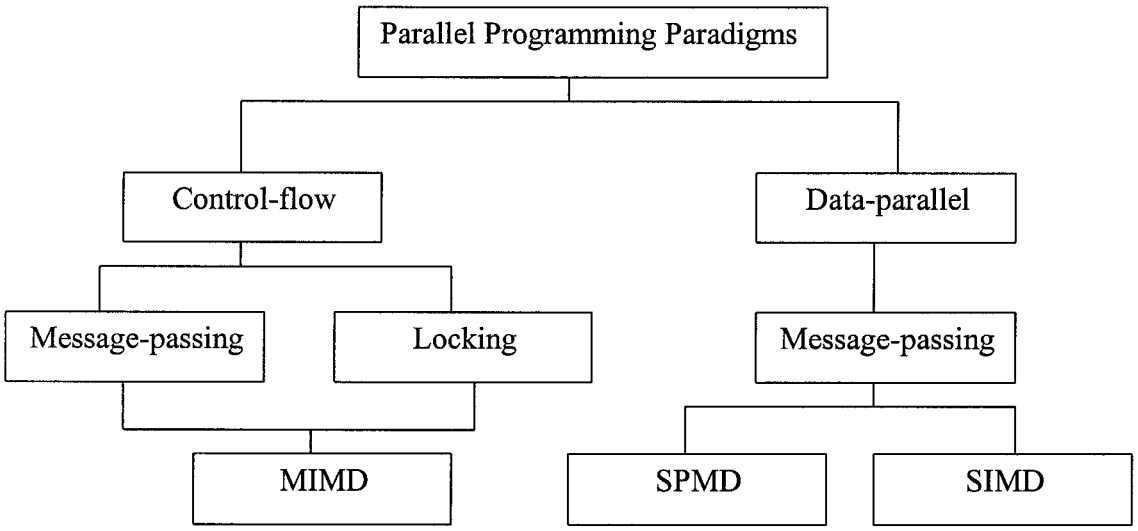


Figure 1.2 The paradigms for parallel programming

1.3 Issues specific to NOW Environments

Architecture wise, a network of workstations (NOW) environment can be classified under the class of MIMD systems. However, because of many challenging issues involved, the area is treated separately under the caption "distributed computing". As a result, NOW environments are harder to program as compared with others in the MIMD class. Some of the issues are as under:

Heterogeneity in processor architectures

A NOW environment may consist of a large number of interconnected machines with processors of different speeds, architectures and data formats. For example, various shared and distributed memory MIMD machines, SIMD and vector machines and sequential workstations are interconnected in a particular NOW.

Heterogeneity in memory structure

The amount of physical memory may be different for different machines. As a result, while estimating the largest problem size that can be solved efficiently on such a platform, we must consider the available memory on each machine and also the memory requirements o the application.

Heterogeneity in network links

This heterogeneity results in different communication costs (in terms of communication time) between different pair of hosts in the network.

Less efficient network links

The low cost links cause:

- high network latency (delays) as compared with the dedicated environments. It makes the communication extremely expensive and restricts the scalability of the system.
- low network bandwidth (range of communication) specially for the Ethernet LANs.

Varying system characteristics

As a result of a multi user environment, the resources (processors and network links) are shared by a number of users in an asynchronous manner. This abrupt use causes the system characteristics like processor workload, network traffic load etc., to be changed frequently.

Priority for workstation ownerships

In NOW environments, the concept of ownership is frequently present when workstations are used for executing parallel applications. Workstation owners do not want their

machines to be overloaded by the execution of parallel applications, or simply want exclusive access to their machine when they are working. Reconfiguration mechanisms are thus required to allow parallel computations to coexist with other applications.

Varying degree of parallelization requirements

A program may need different parallel decomposition and task placement approaches depending upon the underlying machine architecture and other machine specific characteristics.

Different program domains

Parallel applications fall into a number of categories. These programs may have regular or irregular computation and communication, or they may be composed of several subtasks with different processor-machine affinities. Other characteristics, such as the communication-to-computation ratio, could dictate the decision to parallelize and the parallelization used.

Security issues

While using different workstations for running pieces of parallel program codes, security issues are becoming vital to consider. Security is needed in both aspects of host and code.

Host Security

The running code should not be allowed to freely access host sensitive resources as it can interfere in a bad manner with other running applications, access the system memory areas to crash the whole machine, can delete the important files or secretly transfer host sensitive data. All such possibilities could be avoided by running codes in a confined environment that allows only limited access to host system resources.

Code Security

The host should not be allowed to freely access the sensitive data or code areas of the running code. This can be done by performing the computation on encrypted application data and in the same way the results are produced in encrypted form. An alternative way of doing this is to split the computation into small segments, such that no part by itself reveals any useful information about the complete computation.

Reliability issues

The reliability issue comes into focus if we consider the machine failures and also the ownership of the autonomous workstations. Studies [1] proved that the machine failure rate increase linearly with the size of the network. As a practical example, for an environment of 200 workstations (in KFUPM), an average failure rate of 4 machines per day was observed over a period of 30 days.

Additionally, machines are also exposed to sudden shutdowns or crashes by providing an autonomous control to their users.

Software issues

Differences in host operating systems, file systems, database systems and languages available should be masked while dealing with such a heterogeneous environment. Efficient software systems are needed which automate most of the decisions that need to be made in these environments, such as automating the data decomposition, distribution, synchronization and communication for the applications across a wide range of platforms. Some available systems in this class included systems like PVM [12] and MPI [53]. The use of a Java Virtual Machine (JVM) [17] could also be a feasible choice for resolving most of the software issues.

Load Balancing

While running a collection of parallel tasks (comprising a parallel computation) on a set of heterogeneous computers inter-linked in a NOW environment, the mapping of tasks to computers highly affects the overall efficiency of the execution. Studies have proved that a balanced distribution could result in near optimal solutions.

Two major classes of task assignment strategies have been identified as static and dynamic. In a static scheme, all tasks are assigned to the workstations once, prior to run time. As a result, the overhead due to running these scheduling schemes does not affect the overall performance. They, however, fail to take advantage of valuable information about the system and the tasks that become available at run time. In a dynamic scheme, tasks are assigned to workstations at run time. Such schemes can benefit from on-line information about the status of the system. However, they have to be simple, since their running overhead can affect the system performance.

The issue of balancing the load already exists in the dedicated homogeneous environments. In NOW, due to the heterogeneity and other specific issues, the problem becomes much more challenging.

1.4 Problem Statement

The objective of the load balancing is to minimize the total execution time of parallel applications. It aims at improving the performance of parallel computers (loosely coupled or tightly coupled) by balancing the workload of processor automatically during the execution of parallel programs. This aim could be achieved by ensuring the task distribution among available processors with the following two objectives

- Maximizing the concurrency

- Minimizing the communication

The above two objectives are obviously contradicting to each other. The problem is a member of NP-complete set. However, certain heuristics have been proposed in the literature to get near optimal balancing solution.

1.5 Motivations

The growing popularity of NOW environments on both research and commercial levels, the recent advancements in network technologies and the decreasing cost of the hardware involved, are the main motivations behind considering NOW as feasible parallel environments. NOWs seem to compete against the special purpose tightly coupled parallel processing environments due to ever decreasing cost/performance ratio of workstations and availability of very high speed communication technologies.

However the practicality and feasibility of the idea depends much upon the efficient utilization of latent computation power inside such environments. Load balancing is an attempt in this direction.

1.6 Work Objectives

- The main objective of this work is the design and implementation of a load-balancing framework, named JLBS (Java Load Balancing System), for distributed and parallel applications on a local area network of workstations (NOW).

- The framework provides interface mechanisms for the user to write parallel applications as a set of tasks and execute them in a load-balanced manner.
- The framework tests the feasibility of using a Java virtual machine layer for resolving heterogeneity issues and providing users with an easy to use program environment using Java RMI (Remote Method Invocation) and other useful Java features.
- The framework allows to compare the performance of different balancing strategies in NOW environments. Currently, the use of a round robin assignment , a balanced assignment and a random mapping strategy could be implemented.
- Performance characteristics of the JLBS will be demonstrated using generic parallel applications and a real application.

1.7 Thesis Layout

Chapter 2 presents a survey of the literature reviewed in the fields of load balancing and network based parallel computing systems. Chapter 3 gives a complete design specification of JLBS. All the modules and their interactions are described in detail. Chapter 4 discusses an implementation of JLBS based on Remote Method Invocation (RMI) technology of Sun Microsystems. The chapter also talks in detail about the algorithms used to implement different system module. A load balancing heuristic for JLBS is also presented in this chapter. Chapter 5 describes some of the experiments conducted on JLBS and some measured performance figures. Chapter 6 concludes the thesis with comments on future work.

Chapter 2

Related Work

2.1 Introduction

The exceedingly rapid increase in communication network capacity at the local, metropolitan, regional and national and even global level, has enabled a shift in computational paradigms. When the NSFNET was established by the National Science Foundation in 1986 to connect its five Supercomputer Centers, it operated at 56 kbps. The current NSF backbone, the vBNS (very High-Performance Backbone Network Service [56] is in the process of being upgraded from 622 Mbs to 2.4 Gbps, a 40,000 fold improvement in bandwidth over 13 years. The Abilene network [16], established by QWest. Nortel and Cisco for UCAID and the Internet2 project, commenced operation early in 1999 at 2.4 Gbps. The plan is to increase the Abilene network speed to 9.6 Gbps by the end of 1999, an improvement over NSFNET by more than five orders of magnitude. For the LAN environments this advancement is even more progressive, starting from shared medium Ethernet LANs (in kbps) to point-to-point ATM LANs (in Gbps).

The increase in communication capacity far outpaces the rapid increase in the processing power of microprocessor, which has been doubling every 18 months consistently over the same time period. This corresponds to a 500-fold increase in processing power for a

single microprocessor over the same period as the communication capacity has increased by five orders of magnitude.

It is expected that for the next five to 10 years, the growth rate of the capacity of LANs, MANs and WANs will continue to outpace the growth rate of the capacity of individual processors and even the capacity of specially designed high performance parallel architectures. A high performance parallel architecture today consists of a number of processors ranging from a dozen or so to thousands depending upon the type of processors being used. These processors are interconnected through some form of multistage or direct network with individual links having a capacity typically in the range 1 - 5 Gbps. Examples are 1.28 Gbs for the IBM SP2 and 4.8 Gbps for the Cray T3E. Now, the capcity of LANs, MANs and WANs are comparable to the capacity of individual network links internal to high-performance computing platforms. Moreover, the capacity of LANs, MANs and WANs are on a steeper growth path than the capacity of high performance computing platforms.

All these facts motivate the attempts to exploit the potential parallelism intrinsic in the distributed computing environments for running huge computations, which were previously tackled with specially designed high-performance parallel machine architectures. This can produce much higher performance/cost ratios by utilizing the latent and unused computations in such platforms. Studies have shown that for a large percentage of their lifetime, the interconnected machines are used for small tasks (reading e-mail, editing files etc.), thus demonstrating an average idle percentage of at least 90% even during peak hours [4].

Numerous research activities have tried to exploit the computing power of NOW environments by providing platforms for parallel application development. Some of these include PVM [12], based on message passing paradigm and Memaid [35] based on the shared memory paradigm. These meta-computing tools only enable writing programs with using new distributed programming constructs like send and receive messages, remote process execution etc. In fact, these tools provide an abstracted transport layer for having new distributed computing constructs.

2.2 Requirements for parallel computing in NOW

♦ According to Gilbert and Isbelle [15], in order to make full usability of a distributed computing environment, a programming environment should include the following supports.

- *Support for multiple programming paradigms.* Two main classes of programming applicaitons can be distinguished: the message passing programming paradigm, in which processes communicate through message exchanges, and the shared memory paradigm, in which processes communicate by reading and writing share data items.

Although all applications can be written using any of these paradigms, some studies have shown that application performance can be increased by the use of both message passing and shared memory [46] et. al.

- *Support for heterogeneity:* Existing computing environments often include a number of computers with various architectures and performance (parallel machines, workstations etc.), different operating systems, different amount of memories and different nature of links between the workstations; more computing power is then available by resolving these heterogeneity issues.

- *Support for fault tolerance:* In a distributed environment, as the number of nodes achieving a parallel computation increases, possibility of a node failure increases linearly. The failure may be caused by a hardware failure or a restart by the workstation owner. Without provision for tolerance to such failures, it is necessary to restart the entire computation.

- *Support for Load Balancing and Reconfiguration:* While running a parallel computation on a set of NOW workstations, it is desired to put equal loads on each computer. This results in increasing the run efficiency and improvement in process response times. However, due to the dynamic nature of such environments, frequent imbalances results during run time. Also, workstation owners may claim exclusive excess to their machines when start working on them. To cope up with the situation, load balancing and application reconfiguration supports are needed.

♦ According to Mounir et. al. [22], parallel computing on a cluster of workstations in NOW is heavily dependant upon the main issues like *workstation architectures, the*

network protocols, the communication-to-computation ratio of the running processes, the load balancing strategy used and the data partitioning schemes employed.

The paper discusses the strong significance of these factors by deriving an analytical model for NOW environments. The paper then considers the performance of a real life application, a ray-tracing problem, over a cluster of workstations connected by an Ethernet network. The performance results obtained with varying levels of the significant environment variables verify the correctness of their model. The model is then used to find the strengths and weaknesses of an Ethernet cluster of workstations regarding the mentioned factors.

2.2.1 Comparing Load Balancing Strategies

♦ According to Yongbin et. al. [60], for NOW environments, a load balancing can be performed either statically or dynamically.

- ***Static strategies*** [34], [7]] use only the system statistical information like processor processing capabilities, communication costs of interconnections, estimated processing requirements etc. All processes are assigned to the workstation nodes once, prior to the runtime. As a result, the implementation overhead of running these strategies does not affect the overall performance. These strategies, however, fail to take advantage of valuable information about the system and the running processes that become available at run time.

- ***Adaptive or dynamic strategies*** [24], [26], [16] attempts to assign the processes to workstations at run time. Such schemes can benefit from on-line information about the status of the system and therefore thought to be able to further improve system performance However, they have to be simple, since their running overhead can affect the system performance.

♦ Yongbin et. al. [60] use a simulation model to show that both adaptive and static policies improve parallel execution performance (process response time) dramatically and also the performance provided by static policies is not much inferior to that provided by adaptive policies. They show that when overheads are non-negligibly high at heavy system loads, static policies can provide performance more stable and better that that provided by the adaptive policies. The justifications for such, somewhat strange, behavior are as follows:

- No run time overhead and simplicity of static policies
- Possibility of poor scheduling decisions in adaptive policies (due to inaccuracy of system state information), which could cause a continuous transferring of certain processes wasting CPU capacity and thus degrading performance.

2.2.2 Examples of Static strategies in NOW

There has been relatively little work down on static scheduling for heterogeneous NOW as most of the earlier ones assumes homogenous NOWs only.

♦ Zaki et. al. in [23] considers the many aspects of heterogeneity in NOW environments including

- heterogeneous parallel programs which exhibits varying workloads at different portions,
- heterogeneous processors having different speeds,
- heterogeneous memory with different amount of memory available on each host and
- heterogeneous network with different communication costs between processors.

The authors propose a simple yet comprehensive heterogeneous machine model for use in compilation process for a NOW and develop complier algorithms for generating optimal and near-optimal parallel task schedules for load balancing, inter task communication optimizations, network contention and memory heterogeneity. They used a number of application runs including matrix multiply, 2D-Fast Fourier Transform, Cholesky factorization, Spatial price equilibrium modeling etc. for experimentally verifying the improvements. Experiments show that a significant performance improvement is achieved by using the proposed scheduling schemes.

2.2.3 Analysis of Dynamic strategies in NOW

♦ According to Jerrel and Stephen [44], practically a dynamic load balancer implementation can be divided into the following five phases:

- *Load Evaluation:* Some estimate of a computer's load must be provided to first determine that a load imbalance exists. Estimates of the workloads associated with individual tasks must also be maintained to determine which tasks should be transferred to best balance the computation.

- *Profitability Determination:* Once the loads of the computers have been calculated, the presence of a load imbalance can be detected. If the cost of the imbalance exceeds the cost of load balancing, then load balancing should be initiated.

- *Work Transfer Vector Calculation:* Based on the measurements taken in the first phase, the ideal work transfers necessary to balance the computation are calculated.

- *Task Selection:* Tasks are selected for transfer or exchange to best fulfill the vectors provided by the previous step. Task selection is typically constrained by communication locality and task size considerations.

- *Task Migration:* Once selected, tasks are transferred from one computer to another; state and communication integrity must be maintained to ensure algorithmic correctness.

In [44], the authors use above subdivision for their practical implementation and then experimented with different strategies in each phase in a "plug-and-play" fashion. The experimental workstation environment (NOW) they used was assumed to have only homogeneous architectures in terms of processing capacity, no source of external network load and distributed software library routines for implementing communications.

The paper presents two contributions as follows:

- An adaptive (dynamic) scheme based upon heat diffusion for balancing the workload and
- A task selection mechanism that can preserve or improve communication locality of the running tasks.

The proposed techniques were applied to two large-scale industrial applications on a variety of multi-computer platforms. In the process, the work exposes a serious deficiency in current load balancing strategies. It is as follows:

"Nearly all the current balancing work is targeting for applications with tasks having only one computation phase. The workload for this computation phase is used in task distribution. However, for applications having more than one phases of computation in their tasks, with each phase having different load distribution characteristics and strict phase synchronization, these strategies consider a task workload as a sum of workload of all its phases and distribute on the basis of this total workload. The work in [44], demonstrated experimentally the deficiency of such balancing decision in terms of high workstation idle times observed".

In order to resolve this issue, the authors suggested considering task workload as a *vector* instead of a scalar load index value. Each vector component is then represent the load of a

computation phase. The tasks are then distributed by equally distributing workload in each phase, resulting in no or little idle time at synchronization point between phases.

- Zaki et. al. [58] examined the behavior of various dynamic strategies in a heterogeneous NOW environment in the presence of a transient external load. Their main criterion was to compare strategies in the distinct classes of global vs. local (global does a re-mapping of tasks globally while in local an iterative approach is used) and centralized vs. distributed (centralized strategies depends upon a central balancer module for balancing while each workstation is capable of balancing decision in a distributed strategy).

They concluded that different schemes are best for different applications under varying program and system parameters. They thus proposed a customized balancing scheme, selecting appropriate policy on the basis of the environment. In the process, they presented a hybrid compile-time and run-time modeling and decision process, which selects (customize) the best balancing scheme, along with the automatic generation of parallel code (taking an annotated sequential program as input) with calls to a runtime library for load balancing.

Experimental results, using matrix multiplication and TRFD (a benchmark application from [51]), substantiated the approach used by them.

♦ Cermele et. al. [5], compared various strategies in a dynamic environment regarding load balancing activation, load monitoring and distribution decision policies. Their evaluating criterion was to have an efficient strategy, which at the same time guarantees full consistency of the parallel program's execution. They present a practical implementation for the entire class of SPMD regular applications that are based on PVM library for explicit communications. Their contributions include:

- the feasibility verification of dynamically adapting data distribution for SPMD application class and
- presenting an abstract view of the entire reconfiguration process to the programmers (they only need to interact with the activation phase).

The results showed that no one decision policy is proved to be best for all the applications and scenarios used, however the SAI (System Average Imbalance plicy), that looks at the global system imbalance, seems to be the most stable.

2.2.4 Frameworks for dynamic load balancing in NOW

In addition to the above-mentioned comparative studies, there have been a large number of libraries and frameworks developed for exploiting the intrinsic and much cheaper parallelism available in a heterogeneous NOW. Such systems provide sufficient programming support for distributed application development with functions and procedure calls for automating most of the decisions that need to be made in these environments. These may involve the data decomposition, distribution of tasks (in a

balanced manner), synchronization and communication for the applications being developed.

Such frameworks are mostly developed over existing library support for distributed programming. In this way the system could be relaxed from many of the other relevant issues, like heterogeneity, data conversions etc. However, the selection of a distributed platform dictates important features of the environment like programming paradigms presented to the developers etc. The possible choices of this selection can be classified as follows:

- DSM (Distributed Shared Memory) based Libraries
- Message based Libraries

DSM (Distributed Shared Memory) Based Libraries

Linda [28]: Provides a global associatively addressed memory called the *tuple*[1] *space* which can be accessed via atomic operations for adding, removing and reading of tuples. A special *eval* operation, when applied to a tuple of expressions, generates a set of parallel processes, one for each field of the tuple. The tuple space provides an uncoupled communication paradigm, covering communication in space as well as in time: a type remains in tuple space as long as it is not removed, independent of the lifetime of the process that created it.

Although the above features provide some basic facilities relevant with distributed applications, there is a lack of support for modular design, data parallelism and standard

data structures. Moreover, because of the fact that producer and consumer of a tuple are disconnected, the optimization of data transfers is not easily possible. Some more recent works addressed a few of these issues. In particular, the Linda Program Builder (LPB)is a programming environment that supports parallel program construction and parallel data structures at high level.

Agora [27]: Agora allows the specification of shared data types (SDTs) in a Lisp-like syntax. The description o fan SDT consists of the specification of its data structures, the associated set of methods, and an addressing model. Agora provides a set of built-in methods (*create, destroy, read, write, atomic execute*), which can be combined into more complex user-defined methods. SDT objects can be addressed either by using indexing in a linear array scheme, or through the hashing of strings in a hash table.

The built-in methods provide automatic mutual exclusion for access to an SDT object, and the special *atomic execute* method can be used by the programmer to achieve coarser-grain mutual exclusion. Except for these restrictions, processes may access the shared data of an object in parallel. Process control is performed using an event-driven scheme for access to SDT objects. Each process is associated with a queue of activation requests, which can be activated by other processes.

Message based Libraries

MPI [53] : The message passing standard, MPI has been designed to provide inter task communication (based upon message passing) support for distributed applications. In particular, MPI requires every message to be explicitly typed, thus any needed data

---

[1] A generalized data structure

conversion can be applied. In the recently announced version, named MPI-2 [54], some extended facilities, like dynamic process creation, provide support for dynamic resource management in a heterogeneous NOW. Another recent advancement is the initiation of an industry led effort, named Interoperable Message-Passing Interface (IMPI) [43], which enables interaction among different MPI implementations. The main advantage of this approach is that every vendor may implement its own MPI specifically tailored for his machine, while a standardized interface can be used to interconnect the different platforms.

PVM [31]: PVM supports a straight forward but functionally complete message passing model, and is capable of harnessing the combined resources of typically heterogeneous networked computing platforms to deliver high levels of performance and functionality. From the performance point of view, PVM delivers a significant proportion (of the order of 80-90%) of the capacity available from the underlying hardware, operating system, network, and protocols. and it is expected to maintain this characteristic as network and CPU speeds increase, and as protocol and OS software becomes more efficient. The PVM system currently supports an adequate suite of features, and with the integration of recent extensions, will be in a position to cater to a much larger realm of distributed and concurrent applications.

CORBA [40]: The Common Object Request Broker Architecture (CORBA), defined by the Object Management Group (OMG), has recently gained much attention as a supporting platform for portable and interoperable applications. The central part of CORBA, the Object Request Broker (ORB) acts as a mediator between clients asking for a service provided by a server. All interactions is handled by the ORB; thus client and

server are not directly connected and do not need to know details about their counterpart such as the physical location or the implementation language. Using a CORBA specific Interface Definition Language (IDL) a clean interface for an object (compiled for any platform) can be provided for all required services.

CORBA supports heterogeneity in both architecture and software. A CORBA service (Object Code) may be implemented in any language for which an IDL binding exists. This binding particularly specifies the translation of various differing features like data types etc. between the language and the IDL. Currently, languages for which IDL bindings are defined include C, C++, SmallTalk, COBOL, ADA, Java.

Although CORBA provides a clean architecture for coupling different modules and for exploiting heterogeneous systems, it has some weaknesses when taking the requirements of distributed applications in account:

- CORBA does not provide explicit support for data parallelism
- Due to the generality of the ORB, interaction may be expensive
- CORBA does not provide adequate support for high level coordination of tasks

Recently, extensions to CORBA for high performance computing have been discussed in [41].

2.2.5 Existing DLB solutions

♦ Gilbert & Isbelle in [15] describes *Stardust* as a complete parallel programming environment on a network of heterogeneous machines, including both workstations and parallel computers. Stardust provides support for an adaptive balancing of load by

employing a check-pointing mechanism. The key features of the environment are as follows:

- *Support for multiple programming paradigms*: Parallel and distributed applications built with Stardust can communicate both through message-passing and page-based distributed shared memory.

- *Support for check-pointing and load balancing*: The state of an application can easily be saved onto disk. The application can then be restarted on other machines, either to support machine crashes (fault tolerance) or to balance the machines loads

- *Support for data heterogeneity*: Stardust includes mechanisms for automatically converting data between architectures of different types. application programmers only have to specify the type of the data structures they use.

- *Support for heterogeneous process migration*: Unlike most related environments, Stardust implements heterogeneous process migration. It is achieved by using an architecture-independent standard representation of data to save an application check pointed state. This excludes all architecture dependent data (e.g. threads, stacks) from the saved application state. However, the application state saved onto disk does not contain any information that depends on the number of application processes. Thus, the number of processes of an application can change (increase or decrease) at application reconfiguration time.

Performance measures have shown that the load balancing mechanism included in Stardust can lead to a substantial reduction in the application execution time. The execution times of the applications used in the experimentations were reduced by an average factor of 37%. From an implementation point of view Stardust is structured to ease its extension to new operating systems and new types of architectures.

♦ Wang et. al. [10], addresses the two main requirements for NOW environments, named *high performance communication mechanism* and *powerful programming environment*. The contributions of their work are as follows:

- They developed a reduced communication protocol (RCP) for enhancing the inter task communication. RCP is a user level protocol, which avoids the context switching and system scheduling cost.

- They provide a parallel compiling tool for automatically generating an efficient and load balanced parallel code. Their tool uses a specially designed parallel library (over PVM parallel support) and a pre-compiler (current for Ada95 code)to generate the parallel version of a sequential code. The pre-compiler does a data dependency analysis and thus calculates task granularities (for having an optimal data distribution for the tasks). The transformed code is then mixed with the calls of the parallel library to generate the full version of the parallel code.

- They addressed the issue of parallel debugging and provide a practical implementation for debugging a parallel development.

- They used a CRR (Check-pointing and Rollback Recovery) technique to provide fault tolerance in the presence of unanticipated faults.

The load balancing routines were implemented over PVM version 3.0. The system consists of two parts: resource manager and information collector.

- Resource manager resides on some central node and receives all the system based requests (from the running programs) like adding node, deleting node, spawning tasks, ending task, etc.

- Information collector runs on each node and collects local load information and reports it to the resource manager. The load calculation for a host considers CPU utilization, the number of active processes and the relative computing power of the node.

An information collector returns the load information to the resource manager whenever the load varies a lot from the last collection. Resource manager, when receives a new task spawn request, takes the balancing decision on the basis of the current system load status.

A Fractal computation is used to carry out experiments. Experimental results showed considerable improvements as compared to the application runs done without balancing.

♦ Jerrel et. al. [45], describes SCPlib, the Scalable Concurrent Programming Library, basic technology that supports irregular applications both on scalable concurrent hardware machines and heterogeneous networked workstations. The library also includes a framework, which supports heterogeneous communication and file I/O, load balancing, and dynamic granularity control.
The SCPlib programming model is based on a concurrent graph of communicating tasks, called nodes. This model is designed to abstract the mapping of work away from particular computers by encapsulating computation within these nodes.

The load-balancing framework in SCPlib uses a distributed load balancing strategy, which is based on heat diffusion algorithm [14]. The strategy provides a scalable, correct

mechanism for determining how much work should be migrated between computers in a heterogeneous network with transient external workloads.

The algorithm outputs the ideal work transfer values between the computers involved. These transfers are then approximately implemented by having computational nodes transfer across computers. The selection of which nodes to exchange may be guided by both the size of the node and its communication locality with other nodes. During the process, however, if there are too few or too many nodes in the system, granularity management routines are used to increase or decrease the number of nodes by rearranging node boundaries.

Experimental results showed expected improvements in application completion times with load balancing in operation. The improvement is becoming better with the application of granularity control routines.

♦ Bozyigit and Nisar in [3] presented a framework, based on PVM message-passing model, for load balancing distributed and parallel applications in a heterogeneous NOW environment.

The work presents a semi-dynamic strategy with generating balanced host task maps once when they arrived. The dynamic task reconfiguration during runtime is not supported. Thus the framework supports a kind of job entry level balancing. The main features of the framework are as under.

- The task computation requirements are predicted using tasks run information from their past runs.
- The framework uses a periodic collection of system load information.
- The heuristic used for generating a balanced task assignment is based on the minimization of completion time for every parallel task.

To analyze the performance of the proposed balancing heuristic, the authors compare its performance results with those obtained by using random and fixed task map generations. The application test suite used was composed of generic (random) as well as real (matrix multiplication) applications. As expected, the balancing heuristic outperforms the other two with every application.

♦ Bozyigit and Ghouse [13] studied the issues involved in transforming a NOW into a Virtual Distributed Computing System (VDCS) from a practical perspective. They realized the essentiality of two subsystems:

- an adaptive load balancing subsystem, to have efficient runs of parallel computations,
- a fault tolerance subsystem, to have a reliable running environment.

The implementation was based upon Linux operating system running on heterogeneous architectures in a NOW. Both the mentioned system were based on a check point and rollback recovery mechanism (CRR) provided by a process migration subsystem (PMS), built in a previous work by Naseer et. al. [1].

Different strategies regarding collection of host loads, estimating task computation requirements, detection of a load imbalance and the actual load balancing heuristics were tested for suitability in a dynamic load balancing implementation. The load balancing system (LBS) is based on a periodic load balancing policy i.e. a load imbalance is tested at fixed interval of time.

The requirements for a fault tolerance facility are realized as:

- Detect and report machine failures,
- Backup each machine in the existing DCS (cluster of machines in the NOW), and
- Restore/ restart the jobs of failed machines.

On the basis of the above requirements, the components realized in the fault tolerance subsystem (FTS) are,

- Fault Detection Mechanism (FDM)
- Fault Recovery Mechanism (FRM)

The responsibility of FDM is to identify the faulty machines and inform the same to other live workstations whereas for FRM it is to recover the failed applications.

The services provided by FTS are to be utilized by LBS. A workstation that fails and had applications running should be detected and the LBS should be informed. The LBS takes care of the applications by re-allocating them in a load-balanced manner.

A detailed performance analysis of the system has done using a number of hypothetical applications and one real application (matrix multiplication) was presented. Using load balancing the average speedup achieved for different applications was 66% of the theoretical level. Fault tolerance ability has proved to provide a reliable environment.

2.2.6 Use of Java in Load Balancing Systems

All the research presented in the above discussion use low level communication libraries like PVM and MPI. Although these systems offer heterogeneous collaboration of multiple systems in parallel, they showed some limitations towards getting a flexible parallel environment. Some of these are as follows:

- In most of the existing load balancing systems, a migration facility is added outside the kernel space. This requires a modification in the application code e.g. new compilation or linking, in order to make use of the migration facility

- Another drawback of most existing load balancing tools is the missing possibility to influence the load-balancing scheme by the application programmer. Additionally, the tools suffer from the strict separation between system and application-integrated load management. But especially the combination of system and application relevant knowledge can increase the efficiency of load balancing

schemes. For example, it should be possible to avoid useless migrations of shot term processes.

- The biggest problem of exiting load-balancing tools is that migration of running processes is a hard job, even in homogeneous systems. Many efforts have been spent in realizing the problem solution [52]. Some basic approaches are as follows:

  - Condor check-pointing library [50], provides process migrations facility within a heterogeneous environment. However, the actual migrations take place across homogeneous workstations, in terms of architecture and operating system, only. Also, the library lacks calls for inter-process communication, as Condor is mainly a batch processing system.

  - A machine specific compilation for existing architectures is done in advance. A selection of an appropriate binary for the target machine is then required at the time of migration. The approach requires complex maintenance of a pile of binaries as new machines are expected to come at any time.

  - A recent approach to tackle the active migration problem is to interpret the migrating code. The obvious drawback of the approach is the decreased execution speed due to code interpretation rather than direct execution. However the approach gains increasing feasibility as efficient interpreting environments, like Java, are getting popular. Agent-based systems[2] [42] are also a feasible selection for implementing solutions in this class.

Growing popularity and enhancements, in terms of efficiency and other features, in Java platform [17] triggered several proposals for exploiting the features of this popular

---

[2] Agents are autonomous software entities which are able to migrate through a heterogeneous network.

programming environment. Java provides a set of features, which makes its use attractive in heterogeneous environments. Some of these are as follows:

- *Platform Independence:* Once compiled, a Java program is expected to run on every machine for which a Java Virtual Machine (JVM) is available.

- *Common data representation:* In order to provide platform independence Java specifies a common data representation, thus there is no need for data conversion when transmitting data between heterogeneous platforms.

- *Communication features:* Apart from low level socket communication Java also supports high level *Remote Method Invocation* (RMI). Moreover, data transfer is simplified since whole objects can be transmitted using *object serialization* feature.

- *Parallelism support:* Java language has built in support for parallelism using a robust multithreading model.

The main drawback caused by the inclusion of Java, is the *decreased execution efficiency.* Java programs tend to show lower performance as compared to equivalent platform specific environments including C and Fortran. This is mainly due to the additional layer of abstraction (the JVM). However, the enhancements like Just In Time Compilation (JIT) are proved to improve the Java performance significantly.

2.2.7 Existing solutions using Java

Many researches have been performed to test the feasibility of Java as a choice for implementing distributed development environments in a heterogeneous collection of machines. These works can be broadly classified in the following three categories.

- Mobile agent based
- Java-application based

- Java-applet based

2.2.7.1 Mobile agent based

◆ Wolggang et. al. [33], presents a distributed development environment with load management capabilities, named FLASH (Flexible Agent System for Heterogeneous Cluster). The system is based on the concept of mobile agents [42]. Mobile agents are active and autarchic software entities, which can migrate through a heterogeneous cluster system. During migration the agents keep their data, program code, and possibly their execution state. Each agent is also provided with a communication engine (ports) to interact with other agents in the environment.

FLASH uses the facilities of an agent based system, named Voyager [57] from ObjectSpace Inc. Voyager is a completely Java-written system which offers the management structures for mobile objects. The facilities needed from such a system are as follows:

- *Naming, Addressing and Locating Agents:* In an agent based environment, each agent needs a globally unique name for distinction. Beside the name, the location of an agent is an important feature which has to be maintained by the agent system (global addressing scheme). For finding a mobile agent in the system a localization service is necessary which determines the actual address of a desired agent.

- *Control support:* The agent systems have to provide control support for termination, suspension and continuation of agents. These mechanisms are used for having an agent migration control.

- *State export:* The state of an agent consists of the data and execution state. Both have to be restored after migration.

- *Security:* For excluding abuse, both mobile agents and the participated host nodes have to be protected from each other by proper security mechanisms.

In addition to the above-mentioned services, FLASH needs to implement some system specific services, which then enable it to include the actual (transient) load of the underlying hardware as a base for agent migration decisions. But most of the existing agent systems, including Voyager, do not offer services for having enough system information. FLASH addresses this problem by:

- Providing a load information system (LIS), which consists of system specific modules (developed in ANSI C) for collecting load on every host.
- Extending the Voyager agent system with interfaces to the LIS services.

FLASH allows developers to write their application dependant code, which will be embedded inside mobile agent classes as a thread. These agents are then migrated in an adaptive manner in order to have a balanced running environment. The additional benefits are as follows:

- As implemented on Java, FLASH offers an object oriented development. This allows easier parallel development for the application writers.

- FLASH also supports a combination of system and application integrated load balancing. Users are allowed to provide application dependant method code for evaluating a task's remaining workload requirements. Every agent is capable of accessing both system as well as application load information services and thus can implement better balancing decisions.

The paper then discusses the feasibility of agent approach by employing different load balancing strategies for a ray tracing implementation. Results obtained showed that:

- For centralized schemes, where a central module is invoking agent migrations, good results for small clusters but not for increasing cluster sizes. It was then

realized that the communication setup (standard Ethernet) was making a bottleneck as heavily delayed messages could be observed during the application run.

- Decentralized schemes, which allows every agent to migrate itself autonomously, are then chosen for large cluster sizes. Due to heterogeneity, each node load is scaled by multiplying it with a node performance factor. The run observes a heavy agent migration phase, started after few seconds of application initialization, which continues till the system reaches a nearly balanced state. Additional migrations happen only at the end of the application run when dying agents cause unbalanced situations.

2.2.7.2 Java application based

♦ Mathew and Tim in [21], implemented a Java based class framework, named *Ajents*, which provides necessary infrastructure for parallel and distributed applications. Ajents actually allows the use of heterogonous resources available in a generic NOW environment by make use of a "Ajents Server object" on every host. The Ajents server object is responsible for the following tasks:

- Supports the remote object creation on its local hosts. A remote object is a simple class object, for which only a class byte code file is available at some host.
- Supports remote object migrations on any other heterogonous system host[3] with an active Ajents server.
- Support dynamic class loading as needed by the system.
- Support security, authorization and authentication.
- Implements some useful distributed features in the existing Java runtime environment like asynchronous remote method invocation, migration/check-

[3] The hosts is supposed to running the main environment

pointing mechanism for remote objects etc. without the need for having any preprocessing or modification in the existing stub compiler technology.

The main idea is to have Ajents Server running on every host in a heterogeneous environment. User applications written in Java then use Ajents class library calls to use resources on these servers by creating remote objects upon them and invoking their remote methods. Class API for object migration is also provided and a transparent mechanism for tracking down the migrated objects throughout the system is available. However, the current implementation does not provide a system level dynamic load balancing mechanism for the running remote objects. Perhaps it allows the existence of a scheduler object, which selects an appropriate Ajents server from the available ones at the time of remote resource allocation.

Experimental results shows that for coarse-grained applications, the overheads introduced by the Ajents do not adversely affect remote method invocation times. Also, they demonstrate the Ajents capability of migrating relatively large objects several times without significantly impacting the execution time of these coarse grained objects.

2.2.7.3 Java-applet based

Most of the work in this class addresses the issue of global computing, which correspond to a load-sharing environment on a huge cluster of heterogeneous workstations, usually on Internet.

Generally in such systems, the idea is to transfer the computational tasks of some client (Java applets) to some other machine, the server through a web browsing mechanism. Clients have these computations (applets) embedded in their hosted web pages. The server machine could have copies of these computations by downloading them through commonly available web browsers.

♦ The infrastructures introduced in SuperWeb [37] and Javelin [2] brings together three types of entities: clients, brokers and hosts. Clients, who seek computing resources, register with a broker and submit their work in the form of an applet. Hosts, who are willing to donate resources, contact the broker and run the applets. The results are then returned to the respective clients. The role of a host or a client is not fixed. A machine may serve as host when it is idle (e.g. during night hours), while being a client when its owner wants additional computing resources.

One of the most important goals of the above projects is *simplicity*. The design is based on the widely used components: Web browsers and the portable Java. By simply pointing their browser to a known URL of a broker, users automatically make their resources available as hosts. This is achieved by downloading and executing an applet that spawns a small daemon thread that waits and "listens" for tasks (Java Applets) from the broker. However, the use of Applets implies certain limitations due to Applet security concerns. For example, all inter task communication must be routed through the broker and every file access involves network communication (applet can read files from their origin only). Therefore, in general, coarse-grained applications with a high computation to communication ratio are well suited for these environments.

♦ Charlotte [39] is a parallel programming environment, which provides features analogous to distributed threads and shared memory abstractions. The two main components in a Charlotte program are: a manager (i.e. a supervisor task) and one or more workers. The manager process creates an entry in a well-known HTML page at initialization. Users can load and execute the worker code by directing their browsers to this URL.

In order to enable communication between manager and workers, Charlotte requires that a manager run on a host with an HTTP server. This is needed to serve for additional requests for applet classes generated by the first running applet (embedded as applet tag in HTML file). Thus if only one machine with an HTTP server is available, multiple Charlotte managers are needed to run on this single machine, if more than application

needs to be run in the environment. This results in a communication bottleneck and high fault influenced environment. [38] provides a solution for such situations, named *Knitting Factory*.

- Knitting Factory [38] includes an embedded class server to eliminate the need for external HTTP servers for serving applet code. In particular, it serves for the class requests similar to an HTTP server. In addition to this, the infrastructure provides two additional features as follows:
    - A distributed name service to assist users in locating running applications.
    - A direct applet-to-applet communication by providing an modified applet class.

Chapter 3

JLBS Design Details

3.1 Introduction

The main design objective is to provide a framework for building parallel & distributed applications, which can run on NOW in a load balanced manner. Thus the design phase needs to address two basic issues.

- What support application writers could expect from such a system in terms of program development?
- What policies and mechanisms should be adopted at the system level for a load balanced running environment?

The rest of this chapter will explain the following from the perspective of the above two issues.

3.2 Requirements – The Application Developer View

3.2.1 Use of a Standard Programming Model

The developers need a programming model, which could enable them to write applications for accessing the underlying system facilities. That model could be thought of having the following characteristics.

- It should provide syntaxes, which could access the system with appropriate abstractions.
- It should be flexible enough for writing applications with a variety of parallel programming styles.
- It should be portable, widely available and familiar.

The available approaches are as follows:

- A customized language design specially designed for accessing the system facilities is used.
- An existing standard language platform is used with some library extensions specific to the system.

The first approach could provide highly flexible and optimized system performance, but it goes against the intention of degree of acceptance and usage. The later one could be most feasible with some good selection of existing language platforms.

3.2.2 A High Level of Programming Abstraction

A high level of abstraction, which hides most of the architectural details of the underlying system, is desirable due to the following reasons.

- decrease in development cost.
- concentration on the problem at hand
- ease of system independent programming for such system.

However a high level of abstraction may have problems.

- Generic systems with high abstraction support may not take full advantage of system specific features. This deficiency could results in a poor use of such features and thus low performance.
- Some system specific interactions could necessarily be imposed on the developers, in order to have a proper application run.

3.2.3 Flexibility in Style of Programming

Any distributed or parallel application may be viewed as a collection of a number of tasks (objects) communicating with each other and working in a co-operative manner to solve the problem at hand. The heart of the gains involving any distributed or parallel application arise from the presence of concurrency[1]. Concurrent programming is the art of constructing a program containing multiple processes (or tasks) that cooperate in solving a given problem.

Given a problem, decisions have to be made about what and how many tasks the application must contain? What is the best way of parallelizing it? How should the tasks communicate and synchronize? To answer such questions, guidelines are available in the form of programming paradigms for concurrent programming. Some of the commonly used paradigms are:

- **Network of Filters Paradigm**: A filter is a task whose output is a function of its input. In this paradigm, an application is visualized as a network in which vertices represent filters and links represent communication channels. Solutions to many problems can be implemented as a network of filters. Applications having a regular task graph, like binary trees, linear arrays, meshes etc. suit well to this paradigm.

- **Clients and Server Paradigm:** Clients send requests to servers which service these requests. Client processes send messages to a request channel and later receive results from a reply channel. This is by far the most common paradigm used in the field. Examples of clients and servers are graphics servers, authentication servers, Web servers and so on.

- **Heartbeat Paradigm**: This paradigm is characterized by Algorithm 3.1.

[1] Concurrency is defined as a simultaneous run of more than one program segment.

- **Supervisor/Workers Paradigm**: This is a variant of the client-server paradigm. The supervisor breaks up a given problem into smaller sub-problems and puts them on an outstanding queue. The workers pick their work from the queue and put the results in a result queue. The supervisor makes sense out of the results in the results queue. This is a generalization of agenda-driven paradigm.

A parallel programming system is desired to provide a flexible programming approach, allowing the implementation of most of the available paradigms.

Algorithm 3.1: *A Heart Beat Based Algorithm*

```
Repeat
        Put out information from a global variable on incident channels.
        Get information from neighbors into local variables.
        Compute new values using a function F.
        Evaluate a condition C to determine completion.
        If not C Then
                Load global variables to be sent to the neighbors.
        Until C
End
```

Algorithm 3.1: **A Heart Beat Based Algorithm**

3.2.3.1 Approaches For Supporting A Flexible Programming Structure

The main characteristic, which distinguishes different paradigms from each other, is the communication pattern they exhibit among the application tasks. Thus for having a flexible programming style, the main requirement is that every application task could freely choose to communicate with any other task. This requirement is based on the following assumptions.

- Every task has a unique identity in terms of its names, application it belongs to and its instance number in case of multiple running copies of the same task.
- Every task has a channel associated with it, which could be used by any other task for establishing a communication link between them.

Thus every task is associated uniquely with a link record whose format is as follows:

Task Link Record = < Task-ID, Task communication Link>

There could be the following two approaches for having a flexibility in establishing a communication pattern.

- All tasks in the application contain the links for all other tasks i.e. whenever a new task appears, its communication link is supposed to be distributed throughout the application.
- A central task collects links for all tasks in the application i.e. whenever a new task appears, it sends its link record that central task. The central task then provides the task link records to different task on demand or at the start of application.

3.3 Requirements – The System Designer View

The issues involved in providing a balanced running environment are directly related to the environmental constraints. Static environments[2] need balancing computation only once, at the process compile time [7]] while dynamic environments[3] require run-time treatment to cope with the changing conditions [9].

3.3.1 JLBS Environmental Assumptions

The environmental constraints involved in the design can be listed as follows.

- Each parallel application is coded as a collection of executables, each of which corresponds to a separate task of the application.
- Each machine in the network is capable of building a generic virtual machine environment for running the parallel tasks.
- The whole application should be compiled to some generic executable format which can be run anywhere in the underlying system.
- The running processes (executables) have the unpredictable nature in terms of their execution times and inter-process communication requirements.
- Processes will run on a collection of network workstations (NOW) being used by other users as well.
- The underlying environment is totally heterogeneous.
- All the inter process communication is TCP/IP based.
- Processes involved are of relatively high granularity[4].

[2] Static environments assume a priori knowledge about the processes and the system on which they run. Also the environment is assumed to be externally unaffected during the application runs.
[3] Dynamic environments correspond to the situation where running processes may have an unpredictable behavior or the underlying system is being affected by the external entities as well.
[4] High granularity processes are better candidates for running on loosely coupled systems like NOW.

3.4 Balancing Strategy Alternatives

On the basis of the above environmental constraints, *a dynamic load balancing strategy* is needed. However it's not always beneficial to adopt dynamic strategies at their full extent. The non-negligible runtime overheads could adversely affect the system performance.

Compromise between balancing quality and run-time overhead involves two alternatives; periodic and job entry load balancing.

3.4.1 Periodic Load Balancing

In such environments, system will try to maintain a balanced environment by periodically testing the system to see if transferring the running jobs among the workstations is feasible. The overheads involved in this strategy are

- Periodic collection of workstation loads in order to have a consistent system state (environment load monitoring),
- Monitoring the running tasks in order to have estimates of task characteristics for the subsequent runs.
- Task migration (freeze/ transfer / restore)

3.4.2 Job Entry Load Balancing

A partial balancing environment is maintained by mapping the newly arrived jobs to the workstations in a best possible manner. Here the overhead of task migration could be avoided at the expense of low grade balancing. The application and system monitoring are still needed for better solution.

3.5 JLBS Subsystems

In this study a job entry level balancing strategy is adopted. A number of subsystems are intended to form a coherent JLBS. There are seven main subsystems as listed below:

- Load Monitoring Subsystem (LMS)
  This subsystem keeps consistent log of system load by continuously gathering the system loads from the individual hosts.

- Task Dispatcher Subsystem (TDS)
  It does the job of physically transferring the tasks to individual hosts for their execution.

- Application Monitoring/Repository Subsystem (AMRS)
  This subsystem monitors the application running characteristics which are used to predict its future running behaviors.

- Load Balancer Subsystem (LBS)
  The job of this subsystem is to generate ideally an optimal task-host mapping table which will be used later by the central controller subsystem to initiate distributed task executions. The subsystem needs as inputs the current system load state and the task predicted running behavior.

- Service Brokering Subsystem (SBS)
  SBS does the job of globalizing the service access of different subsystem components, by registering and exporting their service interfaces for the rest of the system.

- Central Service Integrator Subsystem (CSIS)
  CSIS acts as a central point for initiating system level activities. It can also be termed as system interface for accessing underlying system facilities. Being a central access point, all the other subsystem are directly or indirectly are under the

control of this subsystem. In other words, the subsystem integrates the activities of the remaining subsystems in every domain.

- User Interaction Subsystem (UIS)
  UIS provides an abstract view of the system services for the user applications. Any service invocation to this subsystem results in the invocation of central controller services which in response carry out the requested functionality.

Figure 3.1 shows a logical view of JLBS subsystems and their interactions with each other.

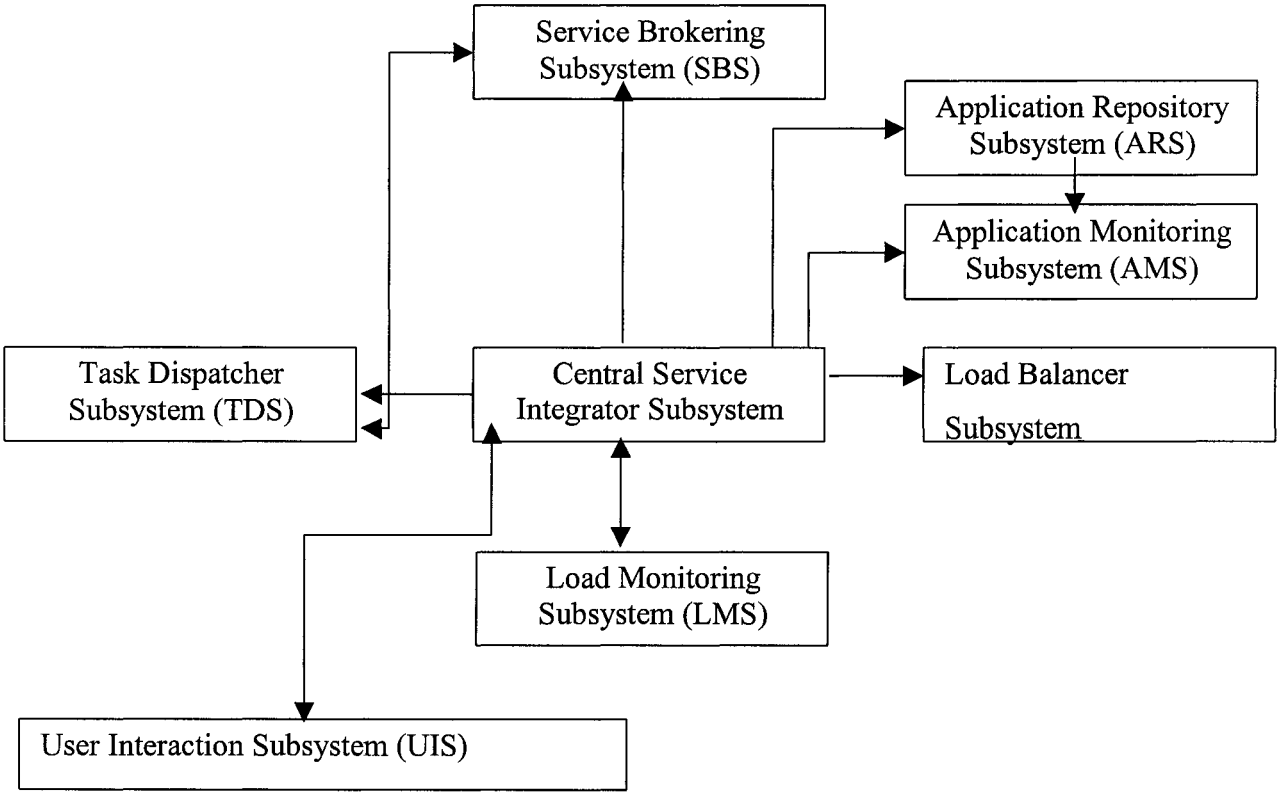


Figure 3.1 Overview of JLBS Subsystems

3.6 Design Alternatives for JLBS Subsystems

Each subsystem in JLBS needs basic system functionality defined by the input it takes, processing it does and the output it generates. This section will discuss in detail different subsystem functionalities and their interactions with the other ones.

3.6.1 Load Monitoring Subsystem

Two basic issues arise in the design of this subsystem:

- What metric defines the load at each host?
- What policy should be adopted for system load collection and maintenance?

3.6.1.1 Load Metric Definition

For defining the load at some host, a frequently used parameter is Load Average [59]. It is defined as the average number of processes in the system. Most platforms directly or indirectly supports system calls for reporting such a load measure.

3.6.1.2 Load Collection Policy

Load collection policies are meant for consolidating system load information by gathering load from all the hosts included. The two issues involved in load collection are as follows::

- Where should the collection take place?
- When should the load information exchange actually take place?

Information Exchange Policy

In a centralized information exchange policy, a dedicated host collects and maintains the system's load information [20], whereas a distributed policy causes all hosts to maintain the system load information individually by exchanging loads among each other. Centralized approaches cause a communication bottleneck and the fully distributed ones are proved to be impractical in large systems due to a linear increase in the communication overhead.

In order to have the advantages of both, a compromised policy, termed as a hybrid or semi-distributed policy [36] is used. The policy views the whole system as a combination of different domains or clusters. Each cluster has its own central load collector and all clusters exchange their cluster load among each other. A probabilistic analysis for the efficiency of such systems is viewed in [47].

Information Exchange Frequency

The information exchange frequency is another important issue to think about. Three choices exist:

- On demand - load information is collected whenever needed. [30]
- Periodic – load information is periodically collected to maintain a consistent system load state at all times [6]
- On state change– host will start exchanging its load information whenever its load state changes by a certain degree [49]

Each of the mentioned strategies have some pros as well as cons. The on-demand information exchange rule minimizes the number of communication messages but postpones the collection of system-wide load information till the time when it is needed, e.g., in a degraded performance case. Thus the main disadvantage is that it results in an extra real time delay.

The periodic rule ensures a consistent view of system-wide load information in real time. This information can be used immediately when any balancing decision is need to be implemented. The problem with this policy is that the existing information may be out of date. The simplicity of periodic rule justifies its adaptation for JLBS environment.

3.6.2 Service Brokering Subsystem

The basic responsibility is to export the services of different system components as well as the application running tasks in order to make them accessible by the rest of the system. This service is implemented as a common service repository, which comprises of the following activities:

- Component Registration - any component, which wants to export its services need to register itself.
- Component Lookup – any component, which wants to access a registered component's service, need to have a lookup for it.
- Component Removal – any registered component, which is going to shutdown itself, needs to be removed from the repository.

The design issues involved are as follows:

- Where should the brokering facility be placed?
- How the system knows about the broker existence on some host?

3.6.2.1 Locating the Broker

As the broker facility is supposed to be used by the whole system, it should be placed at some well-known common central node. However, to make the broker existence more reliable, replications of the repository to multiple nodes could be implemented. Any of such nodes then can act as the broker in case of the active broker node failure. This replication could also be used to reduce the request load on individual broker nodes by distributing the request load through the replicated nodes.

3.6.2.2 Contacting Broker at Runtime

Being a common facility, the broker's existence needs to be well known throughout the system. One way is to allow broadcasting of broker requests. However this method could result in enormous amount of network traffic, which also affects the normal inter task traffic. A more practical approach is to make the local broker address (host name and port number) available throughout the system. The approach needs a little extra work of updating configuration files on every host but contributes in optimizing broker communications at run time.

3.6.3 Task Dispatcher Subsystem

The basic responsibility of a task dispatcher subsystem is to dispatch a task on some specified host for execution. In a job entry level load balancing environments, the functionality is simple to implement. The only issues involved are:

- How to initiate a task dispatching?
- How to resolve the platform heterogeneity?

3.6.3.1 Task Dispatching Initiation

In pure dynamic environments, which use periodic load balancing, for initiating a task dispatching process, usually two alternatives are used:

- Sender Initiated
- Receiver Initiated

Sender initiated refers to the policies in which an overloaded host (sender) [11] is responsible of initiating task dispatching to other eligible hosts. Conversely, in a receiver initiated policy, an under loaded host (receiver) [19] is the one which initiates request to other hosts for incurring their task executions.

For job entry level load balancing, again two alternatives are possible:

- Any host at which new jobs are being submitted for execution can dispatch the tasks. This could be done in environments where every host has the complete system load information and thus capable of initiating a balancing decision.
- A central host to which every other host needs to communicate when they have new jobs to run, can dispatch the tasks. Such strategies are needed when system load information are available only on selected hosts.

Out of the above, JLBS employs the first alternative.

3.6.3.2 Resolving Platform Heterogeneity

The issue of resolving platform heterogeneity can have the following alternatives:

- Multiple compilations for task executables are available at some shared resource and any host can pick its own executable when directed for a task run . This is the method used by PVM or MPI based systems.

- A generic executable format for task executables is used which can be run on any architecture in the environment. In such environments, there is a need to create virtual machine layers on every host. That layer then makes the host capable to run such generic formats. JLBS makes use of this policy.

3.6.4 Application Monitoring/ Repository Subsystem

In dynamic load balancing environments, the load balancing decisions are mainly based on the current system load conditions. However, feedback information from previous runs may also be used for making new, more improved load balancing decisions [25]. This requires some continuous run-time task monitoring facility and a robust task run repository subsystem which allows an easy access (read/write) to such database.
The issues involved in task monitoring are:

- What task characteristics need to be monitored?
- Where to locate the task history repository?
- How a previously run application is identified during its next run?
- What to do with the tasks having no history information i.e. running first time?

3.6.4.1 Task Monitoring Metric

The significant task characteristics metric, which could be used in balancing decisions are:

- Task completion time
- The effective load (execution and communication) on a host caused by this task run.
- Volume of communications with other tasks.
- Static task characteristics: input, output, memory required, etc.

3.6.4.2 Locating Application History Repository

The location of the task data repository for task database should be where the decision process is being taken place. Naturally, this location is dependent upon the adoption of load collection policy as the balancing decision needs to consider the current system load also.

Thus for centralized load collection environments, the task repositories should be maintained at central host. Conversely, for distributed load collection environments, a identical copies of such repositories are needed at every host. For hybrid strategies, every domain should have identical repository copies at its central host.

3.6.4.3 Runtime Identification for Parallel Applications

The application identification is an important design issue due to the following scenarios:

- The users may use same names for different applications
- The users may use different names for the same application

The issue can be resolved if unique identity for each application is generated by the history mechanism at the time of first execution. That unique id could be calculated using some unchanged characteristics of an application, so that the user given name will not change it. Thus the name is not expected to change in the above-mentioned scenarios.

The question arises now is what application characteristics can be used for an application for generating unique id. The choice of such characteristics is an implementation level decision. However, some of the candidates are the sums of sizes of all the executables of an application, extracting part of header information from each executable, and so on.

Let a mapping function f be defined as

$$f: A \times B \longrightarrow C$$

where

A is the set of all possible application names.

B is the distinguishing characteristics of the application and

C is the set of unique application identifiers.

For any application named alpha(α), unique characteristics beta (β) could be extracted. The unique application identifier can then be generated as: *unique identifier* $= f(\alpha, \beta)$.

Whenever an application run is requested, its mapping to a unique application identifier is done transparent to the user. The identifier is then maintained internally in the history repository. The mapping is then registered in the history table database to indicate the availability of the application history data.

Figure 3.2 elucidates the above idea in the context of JLBS. In the figure, history name mapping module generates the unique application identifier. This identifier is then used for locating the application history data. The history table database contains the unique identifiers for previously run applications. It acts as a starting point for searching application history records. A successful search for an application identifier in history table ensures the existence of its history records in the on disk repository.

3.6.4.4 First Time Application Runs

For any application running first time and having no previous history records, an estimate of execution information could be provided by the application developer. One such information is the application's task graph showing a rough communication pattern among different tasks and execution load. For the other task characteristics, meaningful default values could be assumed by the system.

3.6.5 Load Balancer Subsystem

The main purpose of LBS is to generate a host-task mapping to result in a balanced run. The most basic issue involved here is the objectives defining a balanced run environment.

3.6.5.1 Load Balancing Objective

Many studies exist about the objectives of load balancing. There are three main alternatives:

- Balanced resource utilization

- Minimization of response time,
- Minimization of completion time or the maximization of throughput.

The first in the list is about minimizing the idle time of workstations. However, it cannot identify if some workstation is busy with useless work.

Minimization of response time aims at reducing the response time of the parallel application. The problem with this method is in order to minimize response time, some of the better hosts might be more heavily utilized than the others. For example, a host could be used for executing more tasks because of higher communication overheads otherwise. This could lead to a load imbalance among the workstations in the case where these are large number of tasks with high granularity, in an application.

The Minimization of completion time aims at minimizing the completion time of every task. This is a special case of response time minimization.

3.6.6 User Interaction Subsystem

This subsystem provides an interface for the JLBS services to the application developers. The issues involved in the subsystem design are as follows:

- Parallel programming abstraction it supports.
- Level of transparency provided to system services.

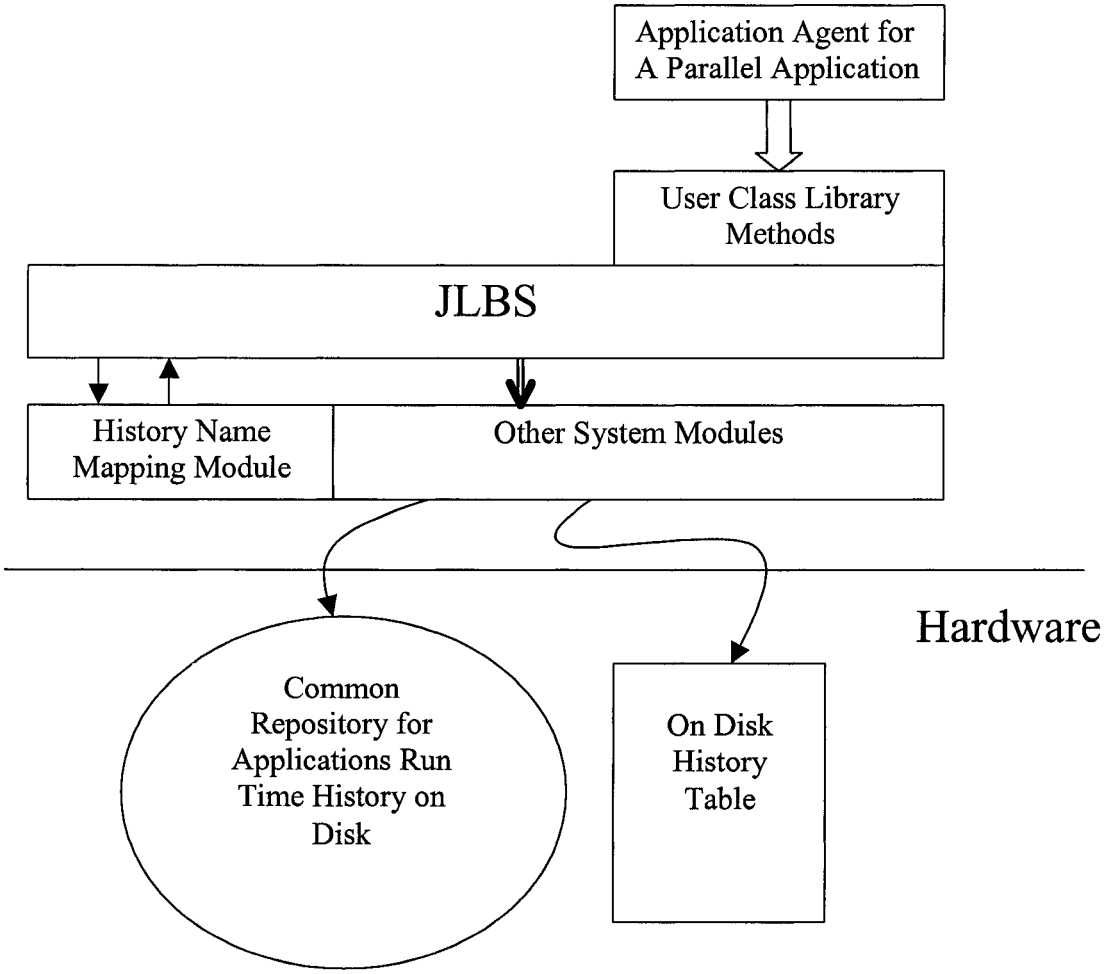


Figure 3.2 Block Diagram for History Mapping Mechanism

3.6.6.2 Parallel Programming Abstraction

The programming abstractions available for parallel programming paradigm are as follows:

- Task/Channel Abstraction
- Message Passing Abstraction
- Shared Memory Abstraction

A collection of concurrent tasks whose number could be varied represent a parallel computation. Each task encapsulates a sequential program code and a local data partition with the addition of an interface to its environment, defined by a set of inports and outports. Task communicates by connecting to their ports with *channels* which can be created or deleted dynamically. The abstraction offers a flexible task mapping to a multi processors without affecting the program semantics.

Message passing abstraction is similar to a task/channel abstraction with the exception of task interacting mechanism. Instead of using channels, tasks interact by sending and receiving messages to and from named tasks. The model allows the dynamic creation of tasks, the execution of multiple tasks per processor. JLBS uses message passing paradigm.

In the *shared memory programming model*, tasks share a common address space, which they read and write asynchronously. Various mechanisms such as locks and semaphores may be used to control access to the shared resources. A big advantage of this model is the avoidance of communication across task boundaries as the notion of data ownership is lacking. Thus, the model may simplify the program development. However, this model does not fit NOW based distributed system's model.

3.6.6.3 Level of Transparency for System Services

A high level access abstraction will possibly hide nearly all the architectural details of the underlying hardware and increases the system utility by providing an easy to program

environment. However, a system with low transparency access to the underlying architecture could be very efficient in the use of available resources.
However, in some particular parallel architectures like NOW, the low transparency access may not be provided at the development level and the system is the responsible for adjusting the execution performance during runtime. The reason is that the execution cost is much dependent on the dynamically changing system properties which cannot be inferred by the programmer during the development stage. The same could be said for applications whose performance varies with different input data or in situations where a particular development is intended to exhibit good performance on a wide range of architectures.

3.7 Summary Of JLBS Design Alternatives

- JLBS will work on a Job Entry Level balancing strategy. The main reason is to avoid excessive overheads, which occur due to active task migrations and other system bookkeeping activities. Such overheads could degrade the system performance to unaccepted performance levels in case of NOW.

- JLBS adopts a hybrid load collection policy based on different load domains. Load is collected on a central basis inside each domain. Domain can share their load information for global balancing decisions. JLBS uses a periodic load collection policy. However, the load collection frequency needs to be decided carefully in order to have a reliable load view with affordable system responsiveness.

- For broker subsystem, JLBS uses a centrally located broker in each domain whose location is supposed to be well known through out the domain.

- JLBS task dispatching will be initiated at a central host in each domain, called domain control host. The task platform dependence is resolved by having task executables of a generic virtual machine environment available at every host in the system.

- Application monitoring/repository subsystem in JLBS will:
  - Locate a complete application run history repository at the central control hosts in each domain. The repositories in different domains are synchronized with each other whenever any one of them changes as a result of some application run.
  - Each running application name is mapped to some unique identity, in order to uniquely identify their history records at runtime.

- The JLBS load balancing subsystem adopts a minimization of completion time based task assignment. The details of the heuristic can be seen in section 3.8

- User interaction subsystem adopts a message passing programming abstraction for developing parallel programs. The programming model has the flexibility for developing parallel applications of any kind except the case in which support dynamic task creation. However, studies have proved that most of the parallel algorithms, which rely on dynamic task creation, can be further refined and converge to algorithms with non-dynamic nature.

  UIS supports both low and high level system access transparency by providing access to both highly abstract and low level system objects.

3.8 JLBS Load Balancing Heuristic

The load balancing heuristic used considers minimization of completion time (or maximization of throughput) for every task assignment, as its objective function. It is based on having a good initial placement of the task. This is because the problem of active process migration is a difficult one and is out of the scope of this study. The heuristic takes the following items as inputs:

the tasks of the parallel application, their completion and communication times

information about the workstation types and their suitability for different types of tasks. For example some of the hosts could be parallel processors themselves.

A measure of the relative distance between the workstations and a measure of the current load on the hosts included in the environment.

Total Cost:

Let an application is represented as A(T,E,C,N) where

$T = \{ t_1, t_2, t_3, t_4, \ldots\ldots, t_n \}$ represents the tasks of the application

$E = \{x_1, x_2, x_3, x_4, \ldots\ldots, x_n\}$ represents their execution times and

$C = \{ c_{ij} \mid$ communication cost between task t_i and $t_j\}$. If the tasks do not communicate then c_{ij} is zero.

Using this information, the total cost of the application becomes.

$$TotalCost = \sum_{i=1}^{n} x_i + \left(\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} \right)$$

Minimization of the above cost has been explored by many researchers. Jack Worlton [32] has studied the limits of parallel computations. He assumed that a parallel program typically consists of repeated instances of synchronization tasks followed by a number of actual computational tasks distributed over the system. He assumed that all the processors had the same capacity. This is not true in our case. Assuming that the difference is not drastic, because of various overheads, the total cost on a parallel system is longer than it would be if they were executed on a single processor, assuming communication is nil.

Parallel Cost:

Let

t_s = syncronization time

t = task execution time

t_0 = task overhead caused by parallel execution

N = number of tasks

P = number of hosts in the system

In a parallel environment, each task requires $(t + t_0)$ time units rather than just t. For N tasks executed on P hosts, the number of parallel steps is the ceiling ratio $\lceil N/P \rceil$. The parallel cost may be approximated as follows

$$T_{(N,P)} = t_s + \left\lceil \frac{N}{P} \right\rceil (t + t_0)$$

Even though this is the case with parallel cost, the response time or the completion time can be reduced due to notion of parallelism, i.e. multiple tasks of the same application execute concurrently and independently.

Let s_j be the speed of host j (normalized with reference to a typical processor) and lj be its load at that instant. Further let lj be a value between 0 and 1. If lj=0 then the processor is fully free and if lj=1 then the processor is fully busy. Then the expected completion cost of a task i on host j, having an average execution time x_i is

$$CompletionTime_{(task_i)} = x_i s_j l_j + k \sum c_{ik} D_{\alpha(i)\alpha(k)}$$

Where

c_{ik} is the communication cost occured when any task k communicates with task i,

$\alpha(i)$ is the host where task i is running,

$\alpha(k)$ is the host where task k is running and

D_{pq} is the distance between host p and host q. This is defined as a function, which returns the time it takes to send a unit amount of data between p and q.

3.9 JLBS Parallel Application Structure

The design requirements at the development stage call for the following application structure.

3.9.1 Application Agent Class

For every application to be run, a special task called the application agent has to be developed by the application developer. The task doesn't take part in implementing the application logic but it is mainly for system specific activities. These may include

- Registering the application with the JLBS.
- Requesting JLBS for task execution as needed.
- Sending control parameters to the running application tasks.
- Establishing a communication pattern among the application task (parallel paradigm).
- Notifying JLBS about the end of the application run (completion time)

Application agent communicates with JLBS using a class library, which provides a high level interface for accessing the underlying system facilities. Developers are expected to write the least required code for implementing an application agent, while utilizing the pre-written methods. Figure 3.3 depicts a developers perspective of JLBS system with application agent is accessing the system with the provided class library method interface.

3.9.2 Application Task Class

A parallel application is built as a collection of task objects to perform the intended computations. At a particular instance during a system run, there may be a certain number of such objects running in the system. The objects perform their computations on their own data sets (MIMD paradigm) based on the applications they were written for. However, they may need to have some interactions with the system environment in order to have a smooth application run.

JLBS also provides class library support for implementing application task objects. The library already includes methods required for certain task based system activities. However, the developers need to provide task specific methods required to implement the application logic.

- Method for setting the task specific data set. This method is invoked by the application agent for supplying the application task data set.

- Method for performing the task computations on its data set. The method is invoked by the system as soon as its data set is received from the application agent.

The control methods, which are already provided in the library class, perform the following.

- Set the task control parameters. These include setting the task's outgoing and incoming communication links. The method is supposed to be called by the application agent task while establishing the application communication pattern.
- Test whether the task is ready to start computation or not. The test will be needed by the system while invoking the task compute method.

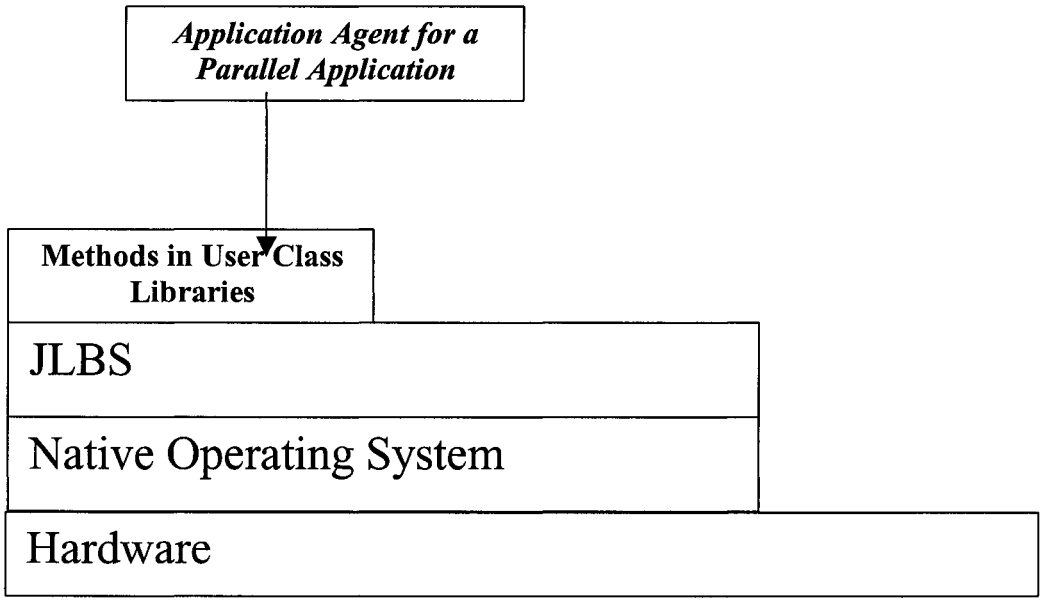


Figure 3.3 A Developers Perspective of JLBS

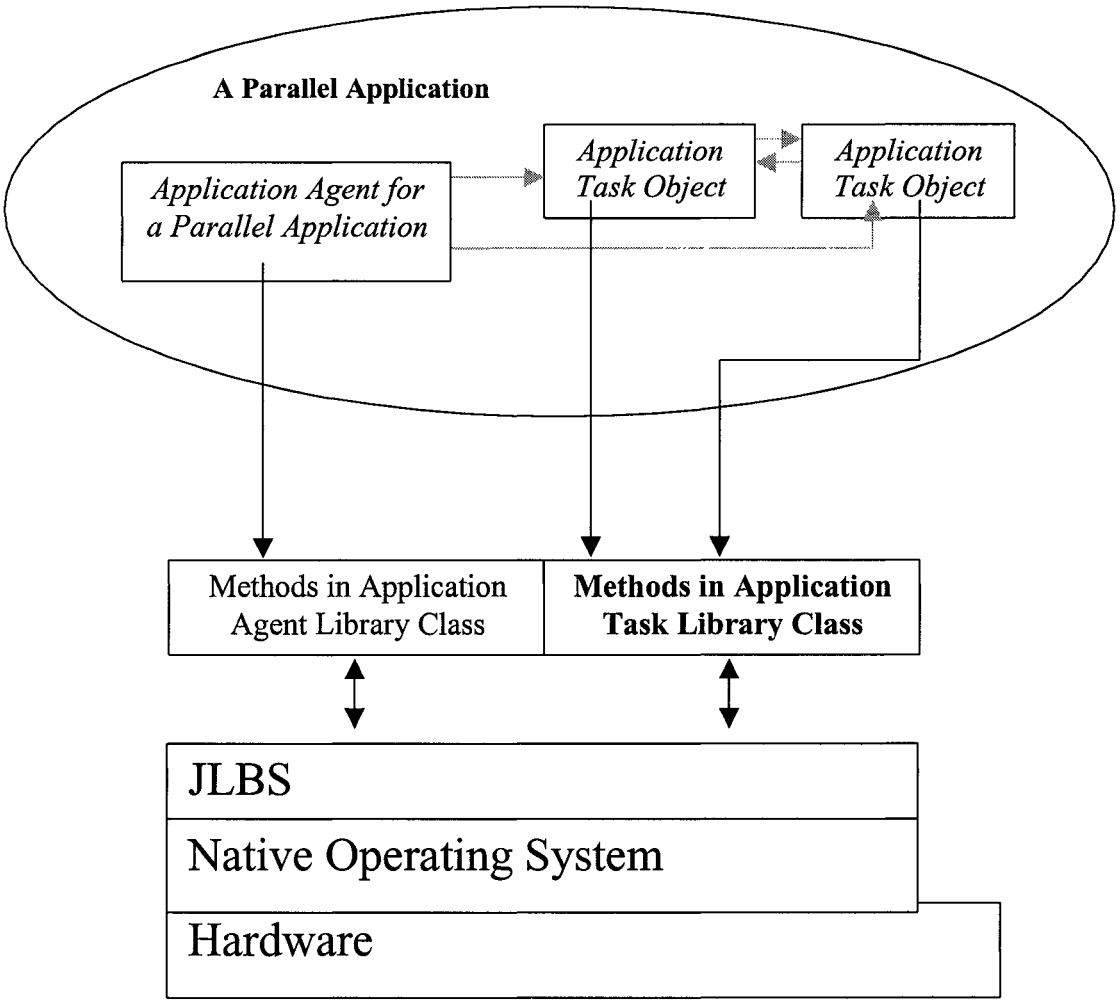


Figure 3.4 A Logical View of A Parallel Application Run

- Send/Receive method to/from a particular task in the application. These methods provide a direct task addressing for send and receive operations.
- Notify the system for the task completion. This method needs to be called at the end of the task compute method. It causes the system to stop certain background activities relevant to this task.

Figure 3.4 represents a logical view for a parallel application run. The application agent and task objects are shown to communicate indirectly by the use of user library methods provided by the JLBS developer support classes.

3.10 JLBS Architectural Design

Forced by a hybrid balancing strategy, the complete system is made up of multiple load collection domains. Each of these domains has identical system objects working for maintaining a balanced domain environment. The domain interacts only when balancing is needed on a global basis. Figure 3.5 presents a pictorial view of the mentioned scenario. Each domain is internally independent with different number of workstations.

The following system objects are identified for each domain, as discussed in the design section.

3.10.1 Broker Object

The object serves as a service exporting facility for the remote objects running in the system. The object serves on per domain basis i.e. each domain has its own broker facility.

3.10.2 Main Controller Object

The main controller objects serve as a service integrator for the remaining subsystems in every domain. It mainly integrates the functionality of load balancer, history management and user interface subsystems.

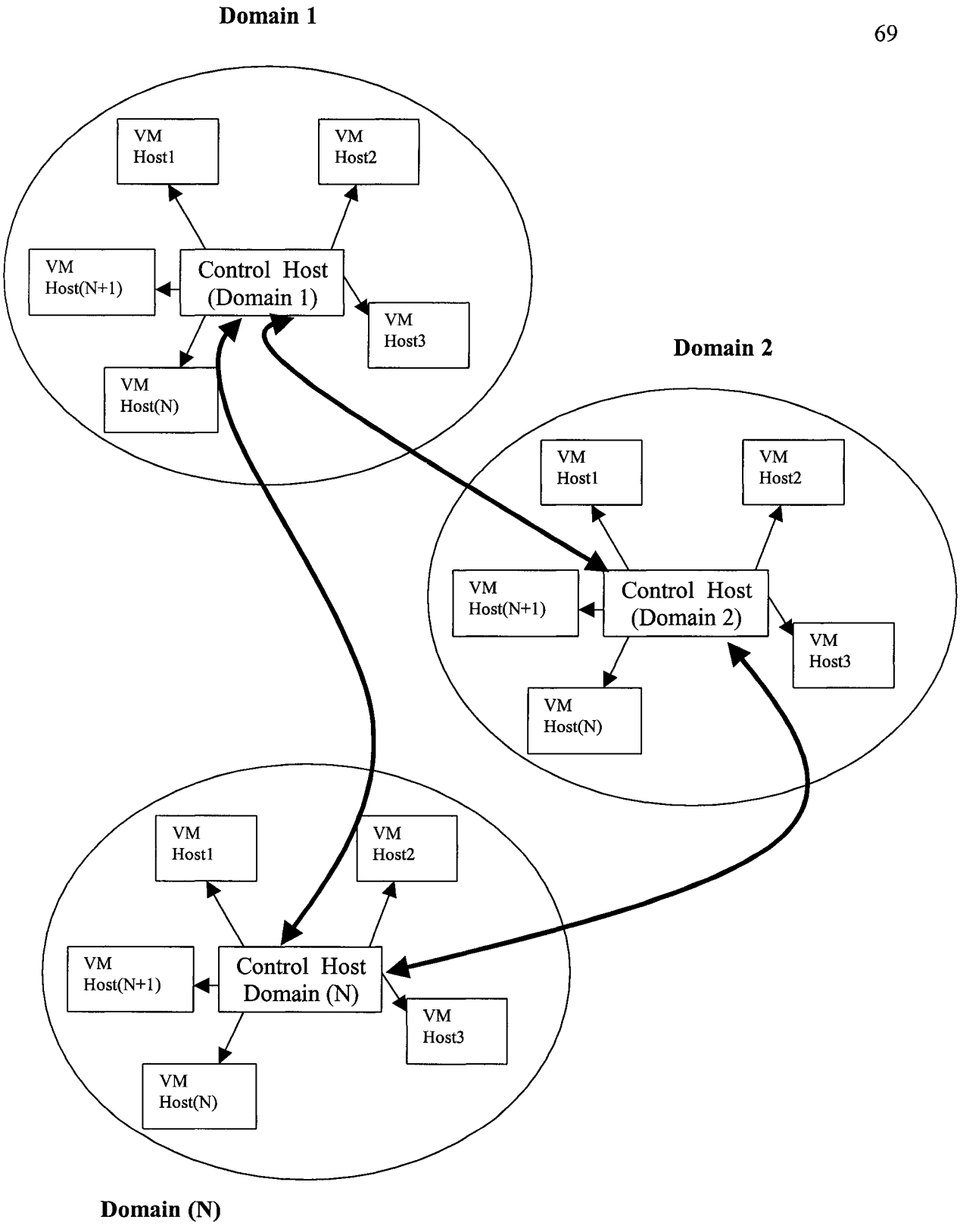


Figure 3.5 An Architectural Overview of JLBS System

3.10.3 Load Collector Object

The load collector object implements the functionality of the load collection subsystem in each domain. The object interacts with its proxies in every domain to have a consistent view of the system load.

3.10.4 History Manager Object

History manager object implements the idea of application run-time repository subsystem. The object resides on control hosts in each domain and directly in each domain and directly accessed by the domain main controller. One object instance is used in each domain. All such object instances are synchronized whenever a change is observed in any domain.

3.10.5 Load Balancer Object

The object implements the services of the load balancer subsystem whose services are directly in use by the main controller object.

3.10.6 Task Agent Object

The object integrates the services of task dispatching and task monitoring subsystem in each domain. Its instances are needed on all hosts in every domain.

3.10.7 Task Monitoring Object

The object is responsible for monitoring the execution of parallel application tasks, which are running on their local hosts. The locally running task agent objects directly access the services of monitoring object to log the running task characteristics. These monitored task characteristics are then reported back to their domain controller on demand.

3.10.8 Load Agent Object

Load agent objects implements part of the logic for load collection subsystem. The object gathers the individual host loads as directed by the domain load collector. Like task agent object, each host in every domain needs an instance for load agent which communicates with their respective load collectors.

3.10.9 System Startup Object

The object can be considered as a single execution entry point for the JLBS system.
It is responsible for providing a single system console to bring the system up and running in a steady state. The object is activated just once during the startup phase of the system.

3.10.10 A Generic Daemon Object

This object implements the system level object interactions, which are needed during the course of a system run. It runs on every host in the system and provides certain low level facilities to other system objects. Some examples of such interactions are as follows:

- During the system startup phase the startup object interacts with a host's local daemon to get the host specific characteristics. It also spawns different system objects on different hosts by interacting with their respective daemons.
- During the startup phase, startup object automatically configures the domain by spawning specific system objects on the most suitable host. At the end, the final domain configuration is also to every host by interacting with its local daemon. This information will later be used in certain system level activities.
- During an application run the domain main controller gets an application task file from the host originating it by interacting with the host local daemon.

3.10.11 Class Library for System Access

Two abstract classes, one for each application agent and application task object are provided. Developers are needed to inherit these objects by adding methods specific to

their applications. The abstract objects are already provided with most of the code concerning with the system activities mainly required. For example, for an application agent the main system activities required are certain bookkeeping tasks at an application initiation and termination, the partitioning of application input data across its running tasks and the receiving of results from all tasks at their completion. For each of these activities, already built methods are provided for the developers.

3.11Detailed Design of JLBS System Objects

This section will describe each system object in detail. Each object description provides a formal definition of the design of the object, interface it provided to others and its interaction with other objects, i.e. protocols for implementing different tasks.

3.11.1 System Startup Object (SSO) Design

SSO object acts as an independent console application having no public interface for the other objects. The object runs only once during the system startup phase and interacts with the generic daemon object on each host to carry out the following:
Implements a system domain configuration supplied by a local domain configuration file.
Instantiate different system objects on particular hosts in each domain by categorizing host capabilities. Table 3.1 gives service interface details for system startup object, as provided by the Java documentation.

System Startup Object
Startup Initialization
During startup, the object evaluates the capabilities of different hosts in each domain. The information is then used later to implement a user specified domain configuration.
Service Interface
The object offers no services to the external environment objects. All methods used are privately accessed by the object itself.

Table 3.1 Service Interface for System Startup Object

3.11.2 Generic Daemon Design

During the course of a system run, there is a need to implement several services locally on each host. These services are occasionally needed by different system objects to carry out their functionalities. Such services are as follows:

- a binary code file
- send an application's task file
- send host speed
- run a JLBS executable
- receives the name for the domain controller host
- receives a list for all control host names.

Table 3.2 gives the service interface details for generic daemon object. The table provides the service names and their descriptions.

The services offered by the daemon object provide the basis for automating the runtime setup of the whole JLBS system from a single console. A system startup object invokes services of daemon objects on individual hosts to get different components of the JLBS system up and running. Daemon object can also be used to dynamically add hosts to a running JLBS environment. This can be done by invoking the objects with different invocation options.

Generic Daemon Object	
Startup Initialization	
The object set up a socket server on a well know port value and goes to wait client requests. The service is provided on a multi threaded basis i.e. the server can server multiple requests at a time. The startup code also checks for the existense of some invocation options. In case of their existence, the object does the requested functionality.	
Service Interface	
Service Name	*Description*
Send binary image of an executable (class) file.	The service is invoked by any interacting host at the time when a binary class file cannot be located locally.
Send an application task file.	The service is called by the central control host in a domain when a parallel program is started from any client host in the domain. The service reads the task file configuration available locally on the client host and send it to the control host.
Send local host speed.	The service is called by the system startup object during the system startup phase. The service reads the host configuration from a local file, compute the host speed and send it to the service client.
Instantiate an object.	The service results in instantiating of a requested object on the local host. The system startup object calls it during the system startup phase.
Receive domain configuration.	The service results in receiving and saving the domain information i.e. the location of broker host, on the local storage. This information will then later be used throughout the system.
Receive system controller list.	The service receives and saves the complete list of controller hosts running in the system. Only controller host calls this service.

Table 3.2 Service Interface for Generic Daemon Object

3.11.3 Main Controller Design

This object stands as the central point of control for each domain. It integrates services of different subsystems in a domain and acts as a central access point for the system facilities i.e. it is responsible for receiving all sort of service requests (initiated from a running parallel application) and direct other parts of the system accordingly.

Table 3.3 gives the service interface details for main controller object. Each entry gives a service name and its description against it. For the main controller object, 12 services have been identified.

<table>
<tr><td colspan="2">Main Controller Object</td></tr>
<tr><td colspan="2">Startup Initialization
The initializations perform during the startup are:
• Exporting its services for the other objects in the domain.
• Instantiation of Load Collector object
• Instantiation of Application History Manger object
Main controller needs the services of the objects instantiated when a balancing decision is made.</td></tr>
<tr><td colspan="2">Service Interface</td></tr>
<tr><td>Service Name</td><td>Description</td></tr>
<tr><td>Start a new parallel application</td><td>The service results in starting the run of a parallel application. The service assumes the following:
• Application's tasks are already compiled to an executable format.
• Application task file has been created.
• The relevant application information is already placed in the system configuration file. These include the application path, application name, number of tasks etc.</td></tr>
</table>

	The service results in loading the history repository of the application in the memory. If the application is running the first time, then the history structures will be populated with some default values.
Ends a running parallel application	The service is used to remove a running application from the memory. The service is called by the controller task of the application at the end of all application's tasks. The result of the service invocation will be: • updation of different history fields with the values from the latest run, • saving of the latest history repository on some permanent storage which could be used for the later runs.
Start a new task	The service is called by the controller task of the parallel application to spawn application's tasks as needed. The task could be spawned by using one of three criteria provided by the load balancer object. • Balanced – the task is placed on the most suitable host to have a balanced run environment, • fixed – the task is assigned to some host depending upon some fixed pattern (like round robin etc.) • random – the task could be assigned to any system host selected randomly.
Controlling the load collection interval	This service controls the load collection frequency in the current domain. The service works as a fine tuning tool to optimize the level of system overhead in different situations.

Collecting the updated system load	Main controller pass this service request directly to the load collector and receives the updated load array from there.
Compute & return the domain load index	The domain load index is a precise representation of collective host load for any domain. The service is responsible to gather and compute the collective load index value inside a domain. This value is then used at the time of balancing when a decision is being made across the domains.
Collecting link references to a supervisor object	During an application run, it is possible to have a supervisor task, which asks for the running statistics of the application. The object is responsible for activating multiple runs of an application and collects the running statistics for building comparison graphs etc.
Collecting call backs of task dispatcher (Task Agent) objects	Distributed task agent objects use this service to submit their callbacks to the domain's central main controller. The service enables the two-way communication between main controller and its corresponding domain task agents.
Return task dispatcher callback list	In each domain, the main controller objects are assumed to have a list of callbacks for the domain task dispatcher objects. These callbacks are then used at the time of actual task dispatching during a balancing process. This service is needed when a balancing is being done across the domains and task dispatcher callbacks from the whole system are needed.
Return last updated system	

load array	For each domain, a consistent host load view is maintained. The service returns the recent load array of the corresponding domain. The service is needed when a balancing decision is being made for a domain.
Saving an application history	The service causes the updated in-memory application history to be flushed on the local storage of the control host. It is called by • the controller task of the running application when the application logically ends. • the domain controller, which runs the application to replicate the application's history throughout the system.
Reporting main controller's aliveness	The service is meant for having a stable system all the time. All objects in the system are consistently checked for their validity by having such a service. For main controller, the aliveness is tested by the distributed task agents, which interact with it occasionally.

Table 3.3 Service Interface for Main Controller Object

3.11.4 Load Collector Design

This object is responsible for gathering load statistics of each domain on a central basis. It collects the individual host loads by periodically communicating with the host load agents. The frequency of this collection is tunable. The object is then accessed by the domain main controller, periodically or on demand, in order to construct a domain load view. Table 3.4 gives the service interface details for load collector object.

<table>
<tr><td colspan="2">Load Collector Object</td></tr>
<tr><td colspan="2">Startup Initialization
The object needs to
• get the object reference of the its domain main controller object by using the lookup service of brokering object.
• export its object reference for other objects in the domain by using binding service of the brokering object</td></tr>
<tr><td colspan="2">Service Interface</td></tr>
<tr><td>Service Name</td><td>Description</td></tr>
<tr><td>Returns the updated system load array</td><td>The service returns the updated load array for the domain hosts. Forced by a design decision, the service is called only from the domain's main controller object.</td></tr>
<tr><td>Controlling the load collection interval</td><td>This service controls the load collection frequency in the current domain. The service works as a fine tuning tool to optimize the level of system overhead in different situations.</td></tr>
<tr><td>Collecting call backs of load agent objects</td><td>Distributed load agents use this service to submit their callbacks to the domain's central load collector. The service enables the two-way communication between load collector and its corresponding domain load agents.</td></tr>
<tr><td>Return load agents callback list</td><td>In each domain, the load controller objects are assumed to have a list of callbacks for the domain load agent objects. These callbacks are then used to collect individual host loads at the time of load collection process.</td></tr>
<tr><td>Reporting load collector's aliveness</td><td>The service is meant for having a stable system all the time. All objects in the system are consistently checked for their validity by having such a service. For load collector, the</td></tr>
</table>

	aliveness is tested by the distributed load agents, which interact with it occasionally.

Table 3.4 Service Interface for Load Collector Object

3.11.5 Load Balancer Design

Load balancer object performs the heart of the load balancing activity. The object generates load task-host maps, which ensures a balanced application run for the current system state. The map is created for the current domain only unless the domain is heavily loaded. In such case, the most lightly loaded domain is selected for a map generation. The object generates host–task maps based on one of three criteria specified below:

- Balanced Task Spawn – task map is generated by using a load balancing heuristic which ensures a balanced application run. The heuristic considers system current load state, host processing capabilities and the execution behavior for the application tasks as inputs.
- Random Task Spawn – task map is generated by randomly assigned task copies to different hosts in the system.
- Fixed Task Spawn – tasks are assigned to different hosts on the basis of some fixed assignment pattern like round robin etc.

Table 3.5 gives the service interface details for load balancer object.

Load Balancer Object	
Startup Initialization The load balancer object does not need to be in an active state all the time. Its services are needed only at the time of balancing decision and are called as indepent function calls. Thus the object doesn't need any initialization.	
Service Interface	
Service Name	*Description*
Fixed generation of task-host map	The service returns a host map table on the basis of some fixed mapping pattern. For example, a round robin strategy of fixed assignments can be implemented.
Random generation of task-host map	The service returns a host map table using a random mapping strategy. In other words tasks are mapped to the hosts by selecting them randomly from the available host pool.
Balanced generation of task-host map.	The service returns a host map table after calculating it using a load balanced heuristic algorithm. The service is performed using current load information and the history repository for task runtime characteristics.

Table 3.5 Details of Service Interface for Load Balancer Object

Figure 3.6 describes the block diagram of this interface.

3.11.6 Component Broker Design

The object is responsible for providing location transparency to the running objects in the JLBS system. Any object, which needs to globalize its access throughout the system, export itself by registering with the broker. A subsequent lookup operation for that object allows others to access its interface in a transparent manner.

The broker object works as a multi-threaded service for the rest of the system. It waits for a broker service request on a well-know port. When a request is received, a new thread is instantiated for servicing and the main thread again waits for any other request.

The functionality assumes the following..

- The location of broker (host address) is well-known to all the hosts in the system.

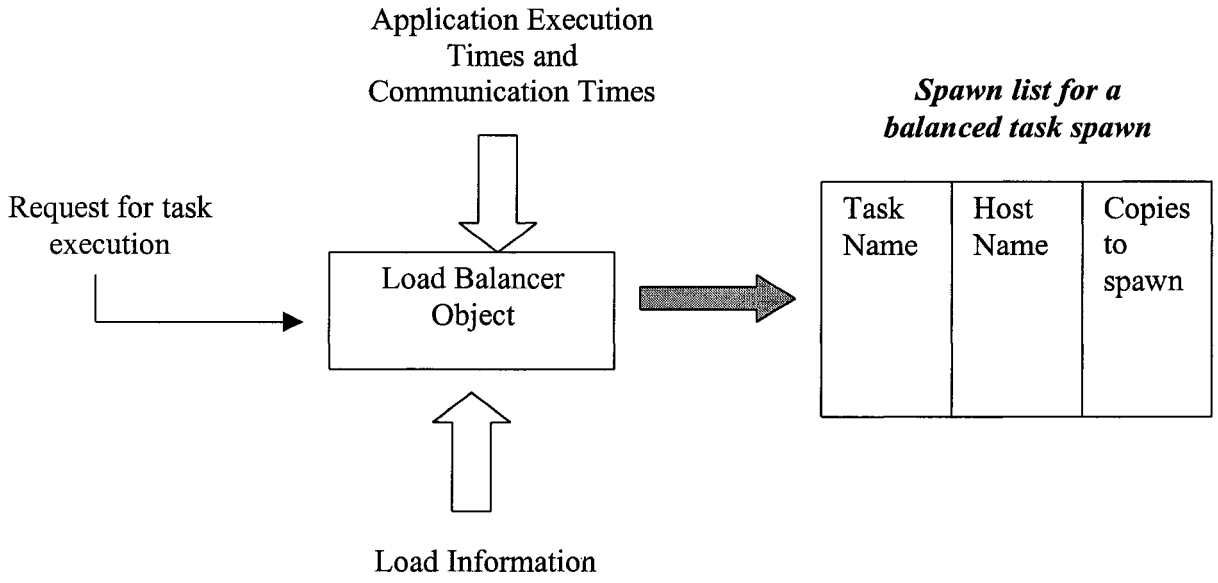


Figure 3.6 Block diagram of the Load Balancer Interface

- A *broker connection* object is available locally for all hosts in the JLBS system. The object behaves as proxy of the broker at every host i.e. it takes broker request from the local system and return the results after passing it transparently to the broker.

Table 3.6 gives the service interface details for component broker object.

Component Broker Object	
Startup Initialization In order to perform the brokering services, the object uses a special repository object which actually implements the storing and retrieving of object references for the registering objects.	
Service Interface	
Service Name	*Description*
Register a remote object.	The service is for registering an object inside the broker. It also binds it with some name supplied by the caller. The name must be unique as it will be used at the time of a lookup service for the registered object.
Lookup for an object's reference handle.	The service returned an object's reference, which was previously registered itself with the broker. The reference then can be used to access the object's services in a consistent and location independent manner.
Remove a registered object from the repository.	After the service is called for any register object, the object is no more available for a lookup service.
Return a list of bind-names for the currently registered servers.	The service is used to have a view of the currently registered objects with their binded names.

Table 3.6 Service Interface for Component Broker Object

3.11.7 Application History Manager Design

Each domain needs to have a copy of application history repository. The repository contains information about the application's task characteristics for all the previous runs. To access the repository cleanly, history manager objects are needed in each domain. Also, after the completion of an application all such objects in the system need to be synchronized in order to have a consistent view of application repository in all the domains.

Table 3.7 gives the service interface details for history manager object.

History Manager Object	
Startup Initialization During initialization, the object looks for the location of the history repository on the local storage. This information will then be used for all the subsequent repository read/ write activities.	
Service Interface	
Service Name	*Description*
Loading an application's history records from the repository	Service is needed at the time of an application startup. The service is responsible for: • Avoiding a duplicate run for any application. This will make the process of running task monitoring much simpler and accurate. • Loading an application characteristic data (either from its history or from its task graph) in the memory
Saving an application's history records to the repository	The service is meant for transferring the application's recent execution statistics to the on-disk history repository. The saved data will be used for the next application run. It will then be overwritten by the new run statistics.
Update application's history	The service is needed at the end of an application. All application related data is then need to be updated with the

	new values in the recent run. The service handler calls all the task monitoring objects which involve in the execution of application's tasks and update the memory record accordingly.
Receive applications spawn tables	The service saves the task-spawn information for an application's task. The table contains a location map for task execution. These tables are made available to the history object as soon as they are retrieved by the main controller from a balancing decision. The information will be needed at the time of updating the task history with the values of the recent run.
Returns an application's history information	The service is needed by the main controller of the local domain while sending a history replication request to other domains. The other domains receive this detailed information and save them in their local repositories.
Removing an application's history from the memory	The service is called after an application finishes and its history is saved in the repository. The service is essential to be carried out in order to allow the subsequent runs for the application being removed.

Table 3.7 Service Interface for History Manager Object

3.11.8 Task Execution Manager Design

The main purpose is to provide an easy to use navigation interface to the task run time monitoring data. While tasks are running in the system, every host maintains a runtime memory log for those tasks. The information is needed to send to the domain control host at the time of task completion where it is finally logged into the task history repository.

Table 3.8 gives the service interface details for Task Execution Manager object.

Task Execution Manger	
Startup Initialization Being a local object at each host, task execution manager doesn't need to export its interface to the other system components. It just initializes its data structures in the memory and then its services will remain at the exposure for managing task monitoring data.	
Service Interface	
Service Name	*Description*
Add node	The service results in the start of monitoring a new task.
Delete node	The service causes the removal of a specified task monitoring records from the running task list.
Update communication data	The service updates the commnunication information related to a specified task. The information include the name of the communicating tasks and the amount of data they communicate.
Get task id list	The service returns a list of all tasks being monitored at some instance.
Get task start time	The service returns the start time value for a specified task.
Get task end time	The service returns the end time value for a specified task.
Get task communication records	The service returns the communication information for a specified task.
Get task average load	The service returns the value in the average load field in a task monitoring record. The value denotes a host load value when the particular task runs on it.
Set task end time	The service put the current value of time in the end time field for a specified task.

Table 3.8 Service Interface for Task Execution Manager Object

3.11.9 Task Agent Design

These objects are the proxies of the application controller on each host. They are meant for the following purposes:

- Physical Task dispatching – A Load balancing activity ultimately requires the physical dislodging of tasks on different hosts. In each domain, the central main controller decides about the task locations to run for having a balanced task mapping. It then uses the task dispatching service provided by the task agent running on each host.
- Task execution monitoring.- For each running task, a continuous monitoring is needed for recording its running behavior. The information is then needed at the task completion to maintain a history log for the running tasks. Task agents are responsible for monitoring the running tasks by deploying an event based task monitoring model.

Table 3.9 gives the service interface details for task agent object.

Task Agent Object	
Startup Initialization During its initialization phase, the object: • Creates a task execution manager for managing the task execution monitoring data. • Deploys an event-driven structure for monitoring the running tasks	
Service Interface	
Service Name	*Description*
Execute a remote task	The service receives a task object from some remote location and runs it on the local host. The service requires the task complete identification and the host from where it is coming. It then downloads the task directly from its origin host.
Return the monitoring data	The service is called when a parallel application ends and its

	task statistics are being logged onto the disk. As a result of this service, the recorded task characteristics are collected and sent to the caller.
Return the list of the tasks running locally	The service is needed when a balancing decision is being made. The information helps the balancing algorithm to compute the communication cost for a certain task.
Record the end time for a running task	The service is called by the locally running tasks to notify the task agent about their completion. As a result, the task cpu consumption time is computed by using its start and end-time values.
Receives the callback for the local load agent.	The service is meant for providing communication between the two local agents running on some host. Task agent later determines the task load characteristics later by the use of load agent services.
Reporting task agent's aliveness	The service is meant for having a stable system all the time. All objects in the system are consistently checked for their validity by having such a service. For load collector, the aliveness is tested by the distributed load agents, which interact with it occasionally.

Table 3.9 Service Interface for Task Agent Object

3.11.10 Load Agent Design

Load agents are the proxies of the load collector on each host of the virtual machine except the control host. The job of a load agent is to collect the host load information on the current host on behalf of the load collector.

Table 3.10 gives the service interface details for load agent object.

<table>
<tr><td colspan="2">Load Agent Object</td></tr>
<tr><td colspan="2">Startup Initialization
The object mainly performs two interactions during the startup.
• The object interacts with the local task agent object on the same host and submits its callback to the task agent. Task agent later uses this call back for getting some relevant host-load characteristic.
The object reads the host configuration specifications by interacting with a configuration reader object. The contents read are then used to calculate the certain meaningful host characteristics.</td></tr>
<tr><td colspan="2">Service Interface</td></tr>
<tr><td>Service Name</td><td>Description</td></tr>
<tr><td>Retrieve current load for the local host.</td><td>The service gathers the load from the local host gathered and returns it to the caller in an agreed upon format. The load definition is system dependent but the format should match with the one needed by the caller.</td></tr>
<tr><td>Reporting load agent's aliveness</td><td>For load agents, the aliveness is tested by the central load collector and helps in declaring them live or dead.</td></tr>
</table>

Table 3.10 Service Interface for Load Agent Object

3.12 Object Interacting Protocols

In JLBS objects interact by the use of messages. Each message sent to some object has certain semantics, according to which the receiver object behaves. A combination of all possible messages an object could receive/send are collectively called the object protocol. However, whenever there is an interaction among two or more objects, an interaction

protocol needs to be defined. It comprises of the set of method calls invoked during the interaction.

The rest of this section describes possible object interactions that take place within JLBS.

3.12.1 Startup Object – Generic System Daemons

The startup object does the job of bringing the whole system up and running by using just a single console. The object assumes the availability of:

- A domain configuration file containing information for a domain setup.
- A generic daemon objects running on all system hosts

The domain record format in the domain configuration file is as follows:

```
<Name of the Domain>
   <Host name 1>
   <Host name 2>
   :
   <Host name N>
```

The startup object creates a list for different domains specified in the configuration file. For each domain, the object communicates with the generic daemon object on each host in the domain list for querying the host speed values. This value is computed from certain useful host characteristics like processor speed, swap size, memory size etc.(see *Algorithm 4.2.1.3)* and could be used as rough estimate for the host processing capability. It then identifies domain control host and simple client hosts in each domain and remotely instantiate respective system objects on each. The startup object also sends system configuration information to each host in existing domains which will be used by different system object at run time. This information includes:

- To every host in each domain, the name of the domain control host is sent. The information is needed by the system objects to directly access their domain control host.
- To every control host in the system, the object sends the complete list of control hosts. A control host in any domain often needs to communicate with other control hosts in the system in order to synchronize to system level information.

Algorithm 3.2 specifies the steps followed. Figure 3.7 gives an interaction diagram for the startup object. The figure shows startup object interacting with two different daemon objects, on a control and an ordinary client host respectively. The messages are shown with arrows and are labeled with the respective comments.

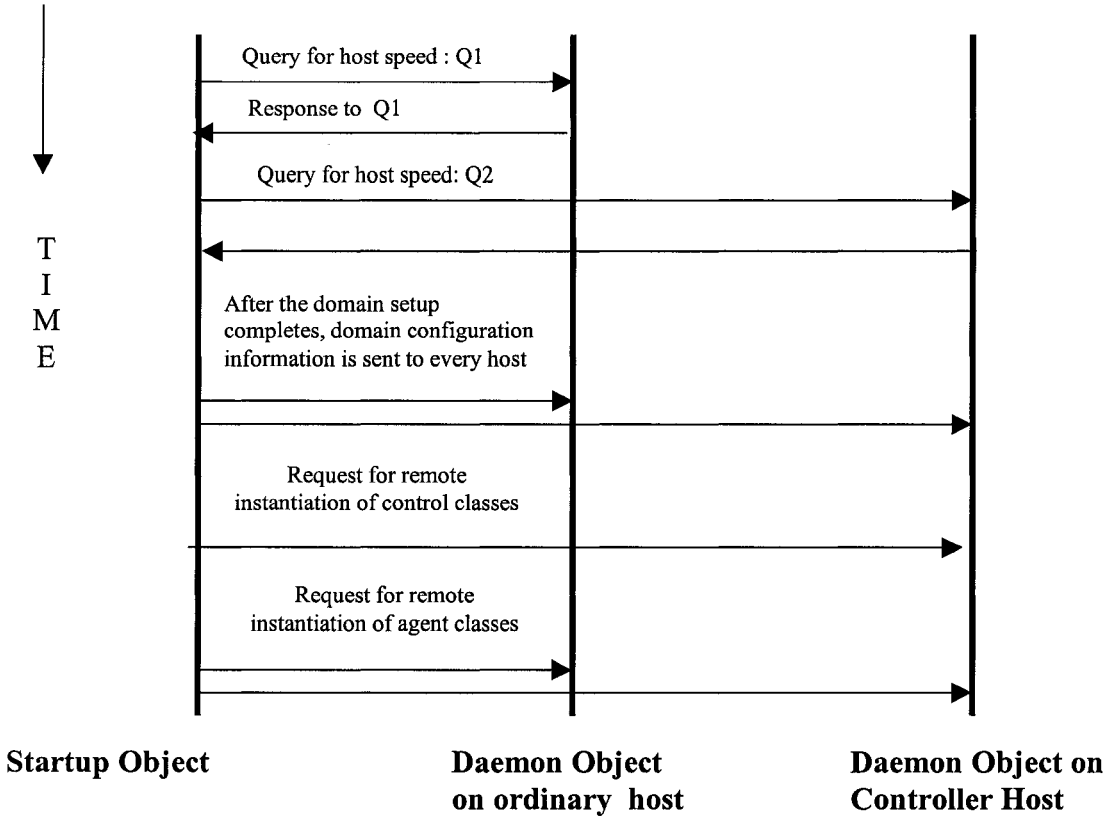


Figure 3.7 Interaction Diagram for Startup and Daemon Objects

3.12.2 Domain Load Collector – Load Agents

This interaction protocol is needed to keep an up to date view of the system load. The load on each host is defined as the amount of work being done by it. The current implementation measures the load as the one, five and fifteen minute averages. This is directly proportional to the average number of processes in the ready and run queues during the last one, five and fifteen minutes.

The load agents periodically collect the load information on individual hosts and send it to their domain load collector on request basis. Load collector then submits the received load to the domain main controller. Figure 3.8 shows the interaction diagram.

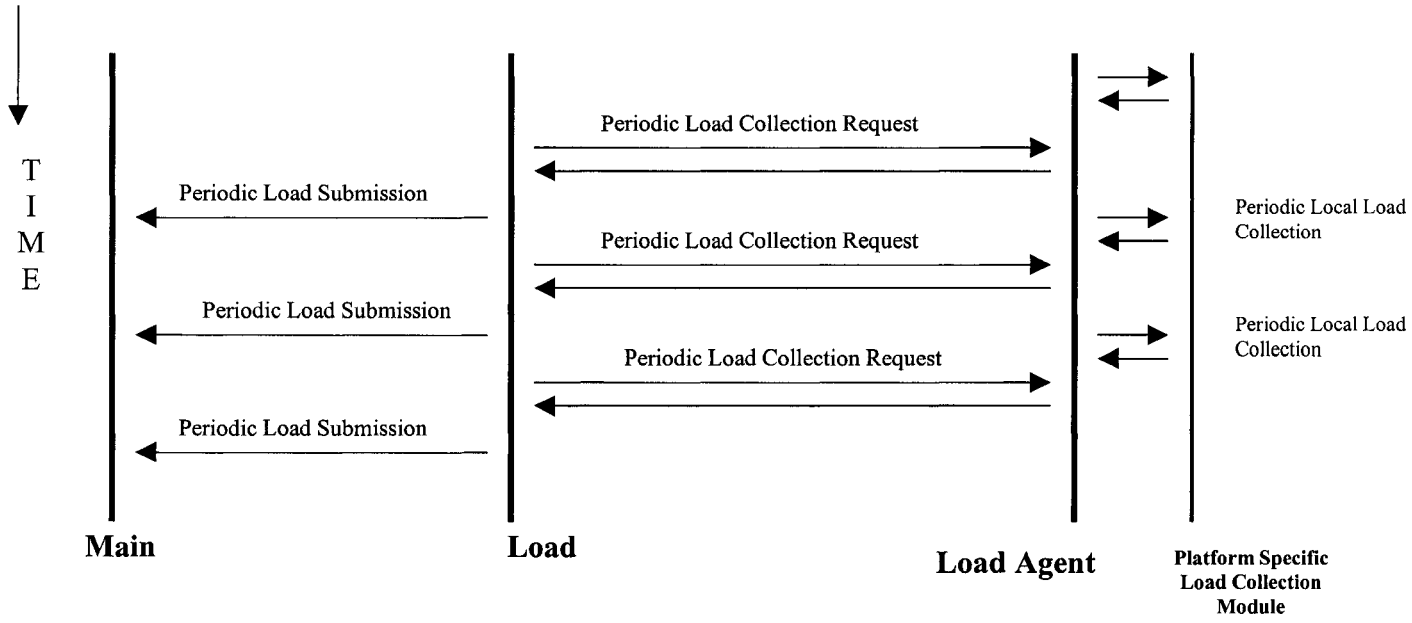


Figure 3.8 Interaction Diagram for Load Collection Protocol

Algorithm 3.2: *SetupSystemEnvironment*
Input: Domain configuration information file
Output: A running system environment
Processing:

```
Read the domain configuration file.
#Determining maximum speed host in each domain
For (each existing domain) Do
   For (each host in the domain) Do
      Send query for determining the host speed.
   End For
   Choose the maximum speed host (MSH) from the received host speed list.
   Declare MSH as the control host [1]in its domain.
   Add MSH name in a system control host list (SCHL)
End For

#Send domain configuration information to all hosts in the system
For (every host in the control host list) Do
   Send the complete control host list to the remote host
EndFor

For (each existing domain) Do
   #sending name of domain controller to each host. This will be used later while
   #dynamically adding hosts in the domain.
   For (each host in the domain) Do
      Send the name of the domain controller host using SCHL.
   End For
EndFor

#Instantiate system classes on different host
For (every host in the control host list) Do
      Remotely instantiate JlbsBroker, JlbsMainController and JlbsLoadCollector
      classes on this host.
      Remotely instantiate agent classes JlbsTaskAgent and JlbsLoadAgent on the
      host.
EndFor

For (each existing domain) Do
   For (each host in the domain) Do
      Remotely instantiate agent classes JlbsTaskAgent and JlbsLoadAgent on the
      host.
   End For
EndFor
```

End Processing

Algorithm 3.2 Setup System Environment

3.12.3 User Library Objects – Main Controller

The method calls provided in user interface class library interact with the main controller in the local domain to carry out the requested activity. The domain main controller may call the services of other environment objects to fulfill the requests raised by the application tasks. The three main services called by the user library objects are:

- An application start request
- A task execution request
- An application end request

The above listed activities are acknowledged by the main controller with the exception of the task execution request. This request is acknowledged directly by the newly created task object. The task object does this by submitting its object pointer to the application object. Figure 3.9 shows the interaction of application object and the main controller.

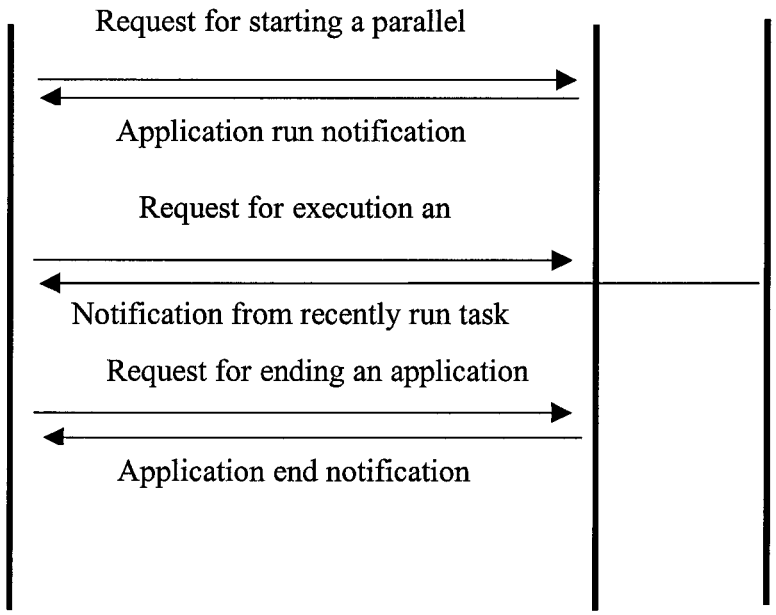


Figure 3.9 Interaction Protocol for Main Control and Application object

3.12.4 Main Controller – Application History Manager

The main controller object in each domain acts as the system entry point for the user applications. For the requests relevant with an application run, the main controller needs to access the application history repository for the domain through the use of an application history manager. The services it invokes are as follows:

- Request for a run environment for a parallel application. It loads the application history information into the memory.
- Request for ending an application run. It will store the recently updated history records for an application in application history repository and removes the in memory history logging structures for that application. It will then synchronize all history repositories residing in the domains.
- Request for reading an application task history. The information is needed at the time of generating a balanced task execution map.
- Request for updating in memory application history records with the values obtained in the recent run.

Figure 3.10 shows the interaction between the history object and the main controller.

3.12.5 Main Controller – Task Agents

This interaction is the key to actual task dispatching. It also results in gathering the task recent execution behaviors from their point of remote runs. The services used by the main controller are as follows.

- Request for running an application task . Task agent directly downloads the task code from its client host.
- Request for run behaviors of the locally executed tasks.

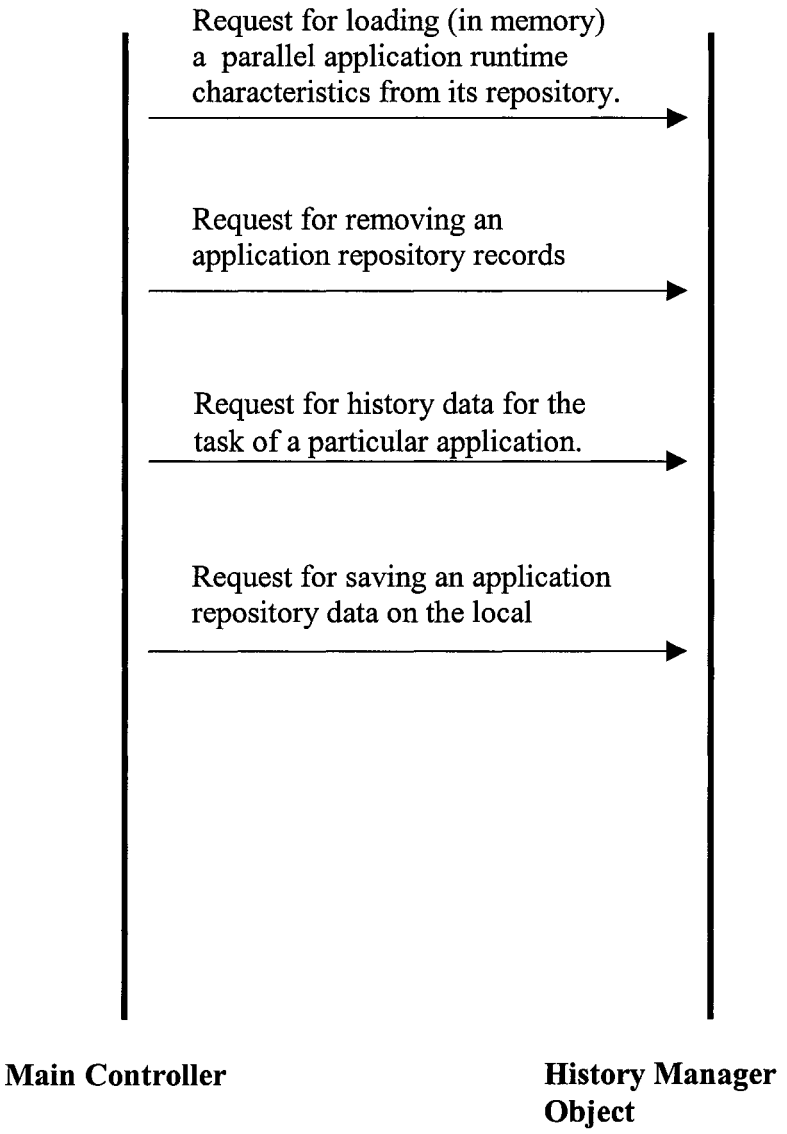


Figure 3.10 Main Control and Application History Object Interactions

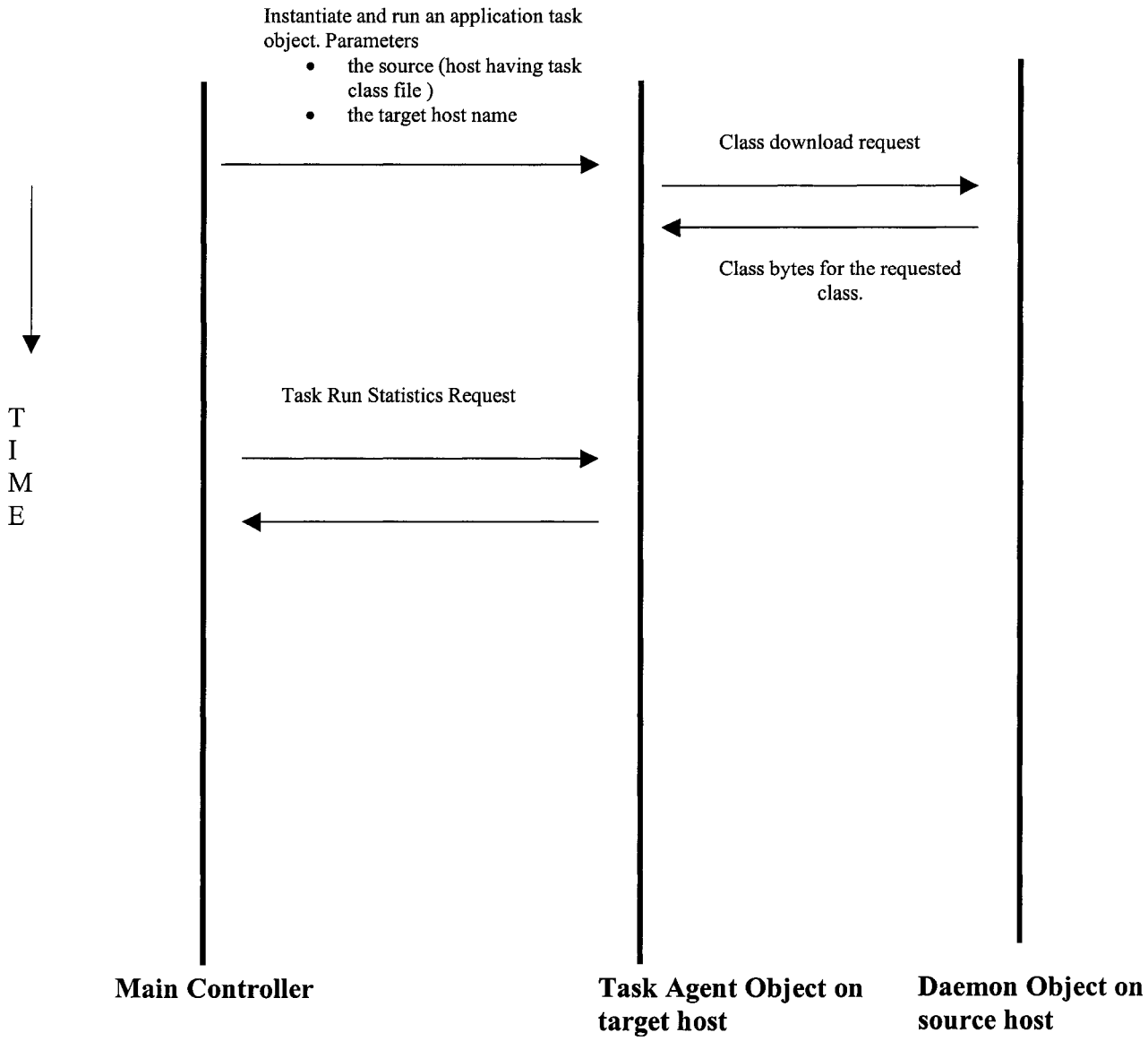


Figure 3.11 Main Control and Task Agent Object Interactions

Figure 3.11 gives an interaction diagram for the main controller and task agent interaction.

3.12.6 Main Controller – Load Balancer

Main controller invokes the services from load balancer object for generating task spawn maps. These maps will then be used for triggering task remote runs at the hosts specified in the map. The services used are as follows:

- Request for a load balanced spawn map.
- Request for a randomly assigned generated map.
- Request for a fixed assignment map.

Figure 3.12 gives an interaction diagram for the this interaction.

3.12.7 Task Agent – Task Execution Manager

This interaction is needed at each host to maintain a log for runtime task behaviors for the locally running tasks. Task agent receives the task execution monitoring data from the underlying system routines and use a local task execution manager object to maintain a log. These logged characteristics will then be passed to the main controller of the local domain on demand. The following services are needed to invoke:

- Request for starting a task monitoring log.
- Request for removing a task monitoring log.
- Request for logging task monitoring characteristics like completion time, run time communication information and so on.
- Request for extracting task monitoring logs. This is needed at the time of sending the task statistic data to the domain controller.

Figure 3.13 gives an interaction diagram for the mentioned interaction.

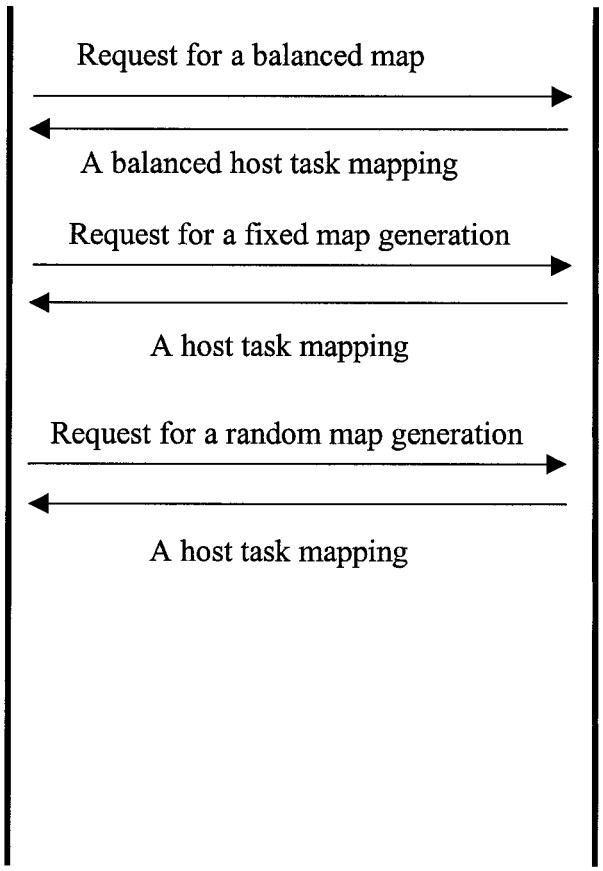


Figure 3.12 Main Control and Load Balancer Object Interactions

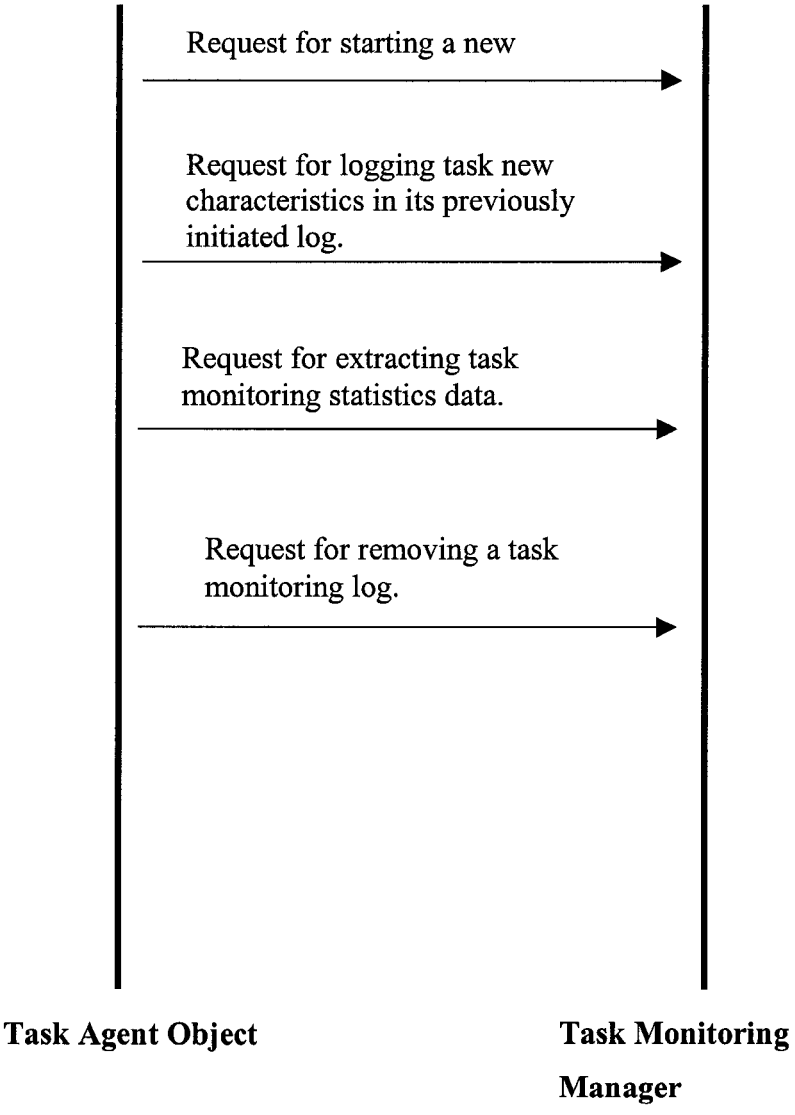


Figure 3.13 Task Agent and Task Monitoring Manager Interactions

3.12.8 A Complete Balancing Scenario

A detailed view of object interaction is shown in Figure 3.14. A complete parallel application run starts from a message from the application agent. Each application run is mainly comprises of three run stages.

- Start a parallel application.
- Executing application tasks.
- Ending a running application

Starting a parallel application

1) Application agent invokes the *start application service* of its local domain main controller.
2) Main controller requests the local history manager to setup the running environment for the requested application.
3) The history manager returns the application history records loaded from the domain history repository.

Executing application tasks.

1. The application agent object generates an execute task request to the local main controller.
2. Main controller requests for a task spawn list to the load balancer object.
3. The load balancer then asks the history manager about the requested task history characteristics.
4. History manager returns the task history behavior.
5. Load balancer then gather the current domain load state by invoking the domain load collector service.
6. Load collector in turn asks for the current host load to all the load agent object in the local domain.

7) Load agents result with the current host load records. The status records are supposed to have the host speed index, showing the host computing capability.
8) Load balancer generates a task spawn list chart to the domain main controller.

9) For each entry in the spawn list, main controller communicates with the task agent object on the remote host for remotely dispatching the task there.
10) Task agent as a result starts a monitoring log for the task run characteristics in task monitoring object.
11) Task agent instantiate a copy of the user object for the requested task on the local host.
12) A background system thread object subsequently reports about the task run time communication behavior to the task agent object.
13) Task agent continually logs the received behavior to the task monitoring object.

Ending a running application

1) Application agent generates an end application request to its domain main controller.
2) Main controller in turn starts calling all task agents which are involved in running the tasks of this application.
3) Task agent calls the local task monitoring object for extracting the recent task monitoring records.
4) Main controller after receiving the recent task monitoring data, updates the history
5) Main controller then calls history object to remove the application history records from memory.

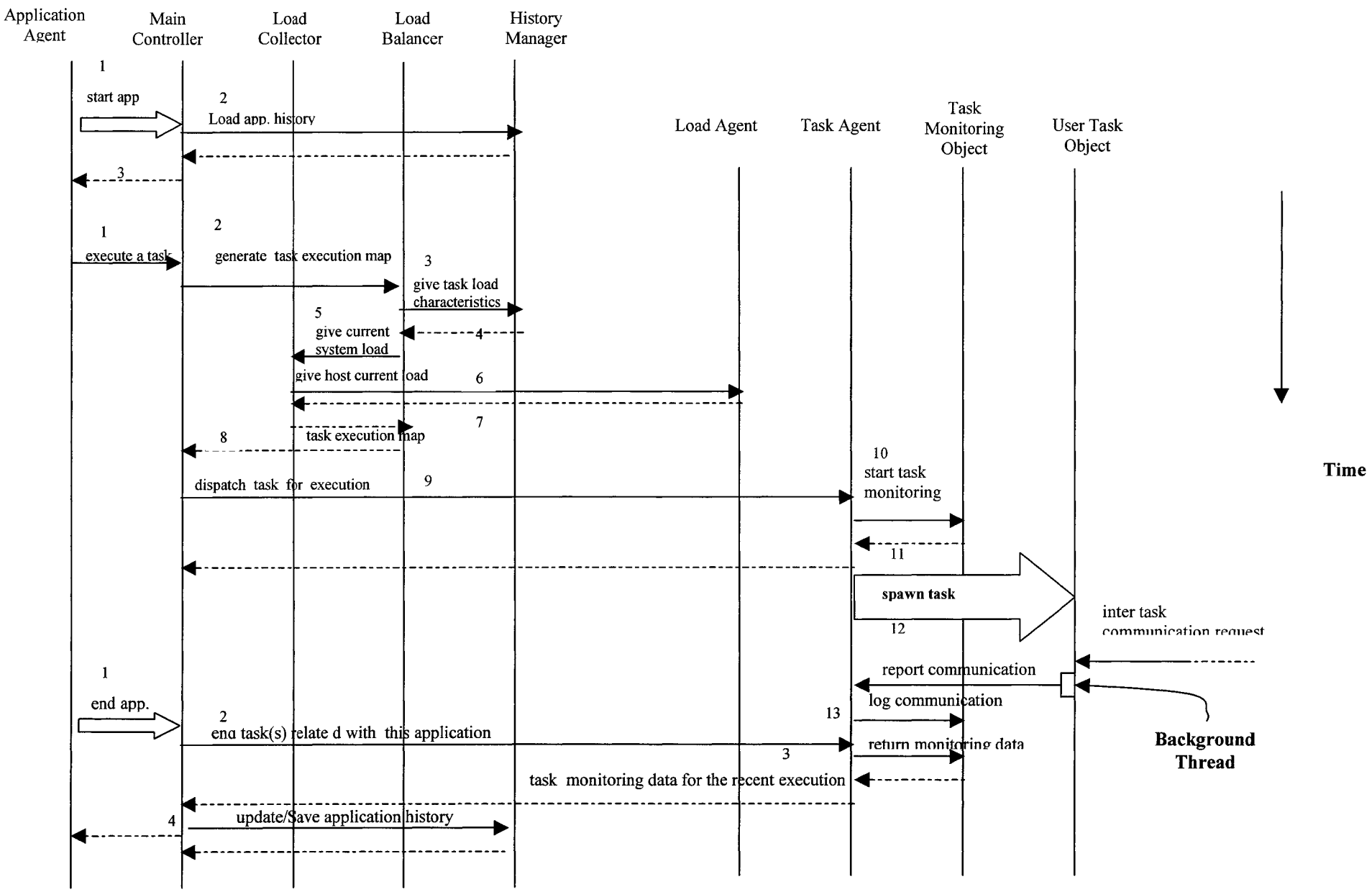


Figure 3.14 System Interaction Diagram for a complete application run

Chapter 4

Implementation Details

4.1 Introduction

This chapter provides a detailed discussion on the implementation aspects of JLBS. The complete system is implemented as a system of interacting objects distributed throughout the network. The chapter will look into the internal working details of these objects and also the inherent overheads involved in their workings.

Each object has a well-defined service interface and also an interacting protocol with others to carry out certain system behavior. Details of the algorithms involved in implementing certain system behavior will also be taken into account.

4.2 JLBS Working Environment

Java virtual machine environment (JVM) is the implementation environment of JLBS. One of the compelling reasons for implementing distributed and parallel applications in Java is that it frees programmers from being concerned with problems due to differences in architecture that traditionally plague heterogeneous distributed and parallel developments.

4.2.1 Programming in Java

Java allows writing programs which could be compiled into a platform neutral form, known as Java byte code. The byte code can be executed on any machine that implements a JVM. The JVM specifications are standardized by Sun Microsystems and open for any

machine vendor. The issue of architecture difference is resolved by a JVM abstraction, over the native operating system.

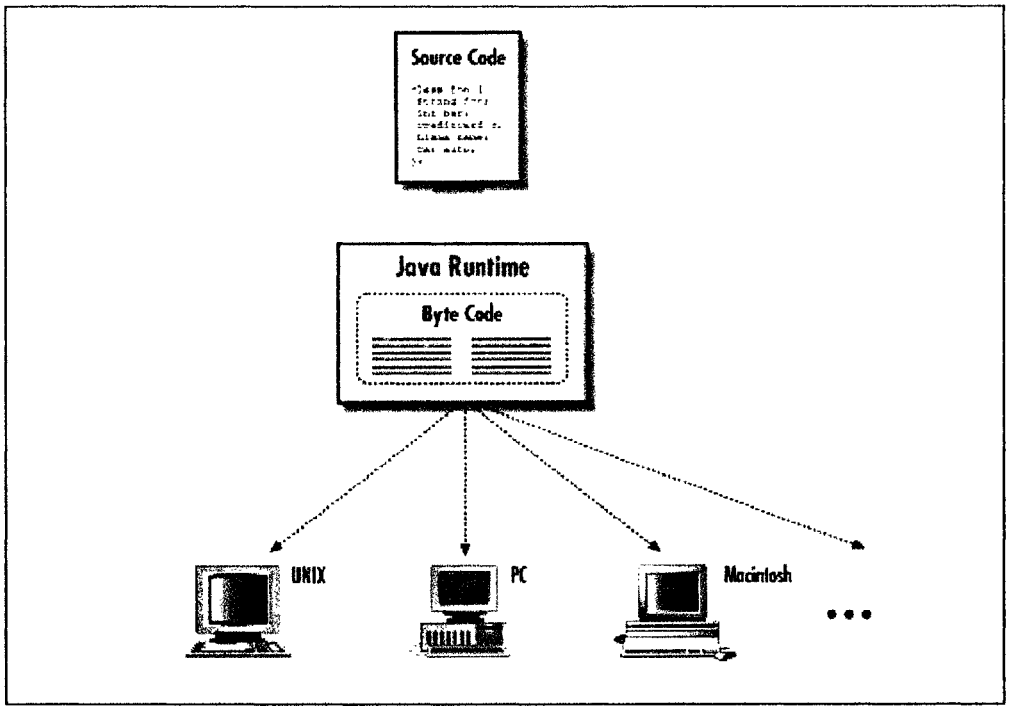


Figure 4.1. Execution of a Java byte code in a JVM Environment

This is done by translating a byte code to native machine executions during runtime. A JVM interpreter actively performs this translation. This may naturally slow down the overall execution performance. Figure 4.2 shows the running environment of a Java byte code.

The degraded execution speed of Java code was considered as a significant issue. Among the solutions, the most famous is the Just In Time (JIT) compilation technology. It does a complete byte code to native code translation on fly and then used the translated native code. Table 4.1 provides a brief look of the available approaches on the code execution speed attached to different statement semantics [61].

Operation	*Java*	*Java + JIT*	*Native code*
Local variable assignment	67	10	5
Instance variable assignment	280	14	5
Method call	541	40	40
Synchronous method call	1767	1215	2053
Object creation	2189	2250	1701
Array creation	178	14	13

Table 4.1 Time taken in nanoseconds to perform various operations

4.2.2 Distributed Programming in Java

Java also provides strong support to distributed applications. The support ranges from the basic use of standard socket abstraction to the much higher level of network abstractions like the use of URL (Universal Resource Locator) and Remote Method Invocation (RMI) technology. Appendix A provides details for various Java aspects for distributed programming.

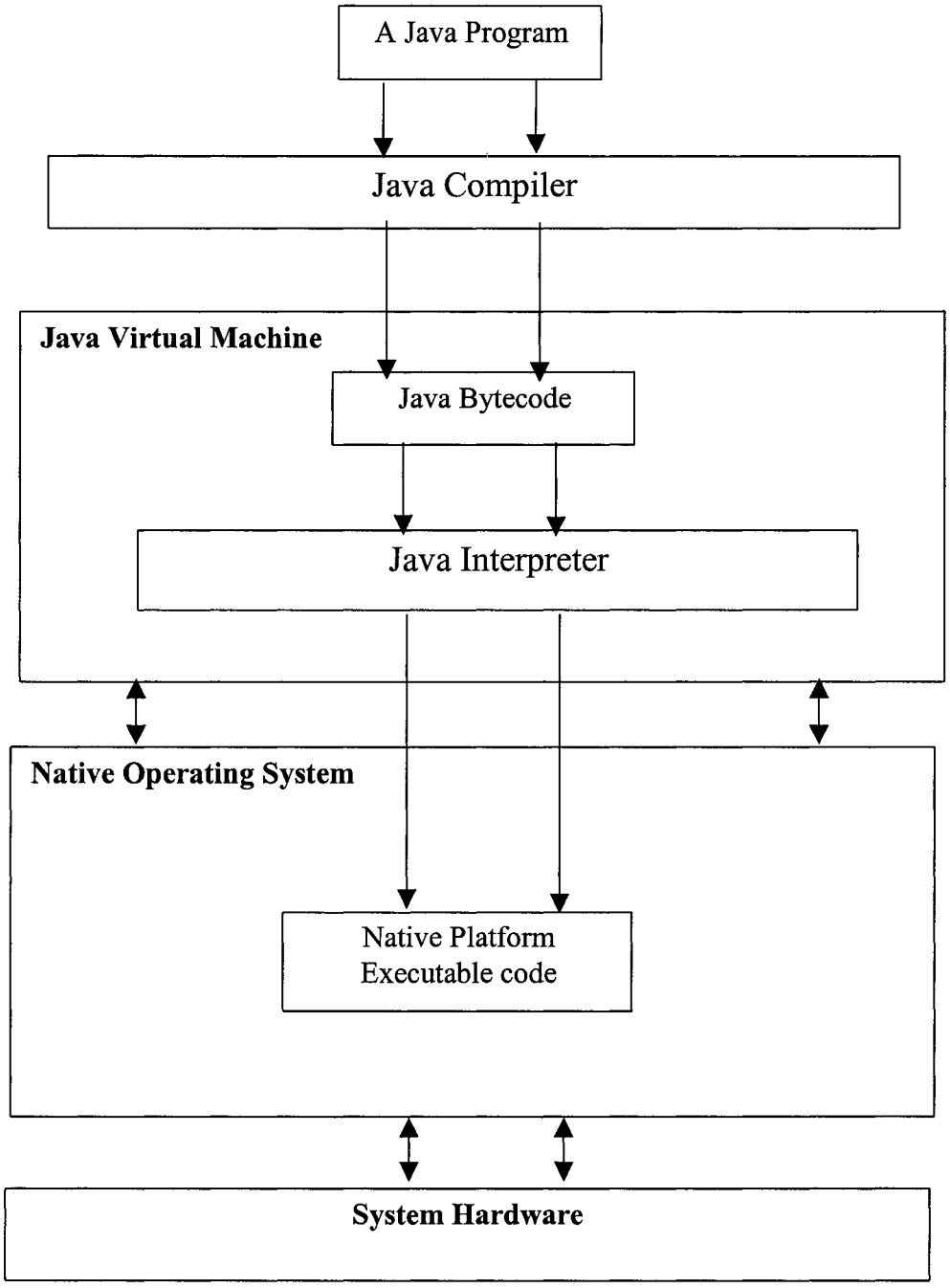


Figure 4.2. Components of Java Runtime Environment

4.2.3 The JLBS Implementation Platform

JLBS uses the Java RMI abstraction to implement its distributed object environment. The Java RMI provides the ability to create and use remote objects. In other words, it enables a client running on one machine, say www.ora.com, to invoke a method in server running on a different machine, say sunsite.unc.edu [48].

Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object. All the peculiarities involved in resolving such references are completely transparent to the user.

4.2.3.1 Details Of RMI Working

Referencing Remote Objects

The mechanism of calling objects on different machines is made possible by the use of ***remote object references***. Like a simple local object reference, a remote reference also represents an object by having the necessary details for referencing it. The only difference is that the object referred to by a remote reference may reside at a remote location.

Thus, a remote reference actually contains the transport details for its remote object, like the address of the machine on which it resides and the communication port value it uses to acknowledge the client calls.

Unlike a local object reference, a remote reference cannot be created locally on the client machine. It should come from some remote location where its respective remote object resides. This could be done by adopting any one of the following three approaches:

- A client object working with a remote reference of any server object can send the server's remote reference to any other remote client object.
- A remote reference is supplied as a method parameter to the client object.
- A registry mechanism is used to register remote objects on some machine which can be contacted on a well know port address by any client to get a copy of the

remote references it has[1]. Figure 4.3 shows the lookup mechanism used to obtain a remote object reference. On the server machine, two objects are at work: RMI Registry and RMI Server Object. The former only provides a copy of the remote reference to the RMI Server that has already been registered into it. The supplied remote reference then can be used by the client to invoke any remote method on the server object.

Thus the remote reference mechanism realizes an additional abstraction layer above the transport layer. This layer is mainly responsible for understanding what a particular remote reference means. In other words, it translates the local client requests referring to a remote reference to the location actually referred to by the reference. This layered design opens the possibility for building different semantics for a remote reference request. For example, a remote reference request could be translated to an object residing on the same machine or on a distant (remote) machine. It could also be used to build a semantic of multicast object referencing i.e. a remote reference could be forwarded to multiple objects residing on different machines. Figure 4.4 is making the idea clear by showing three different semantics of referencing remote references.

- [1] The current implementation of RMI also provides this registration support as a library class, named ***java.rmi.registry.Registry.***

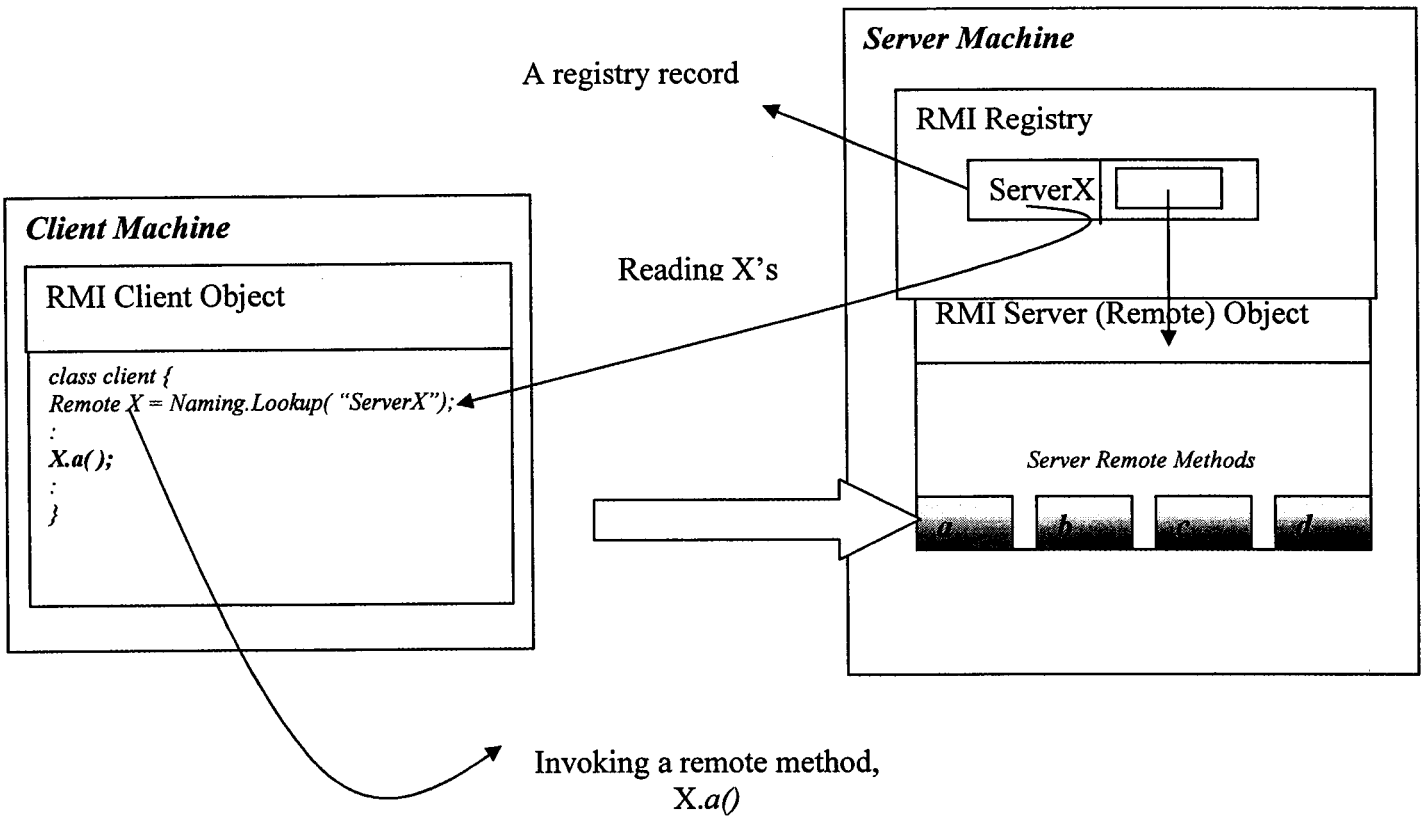


Figure 4.3. Referencing an RMI Object

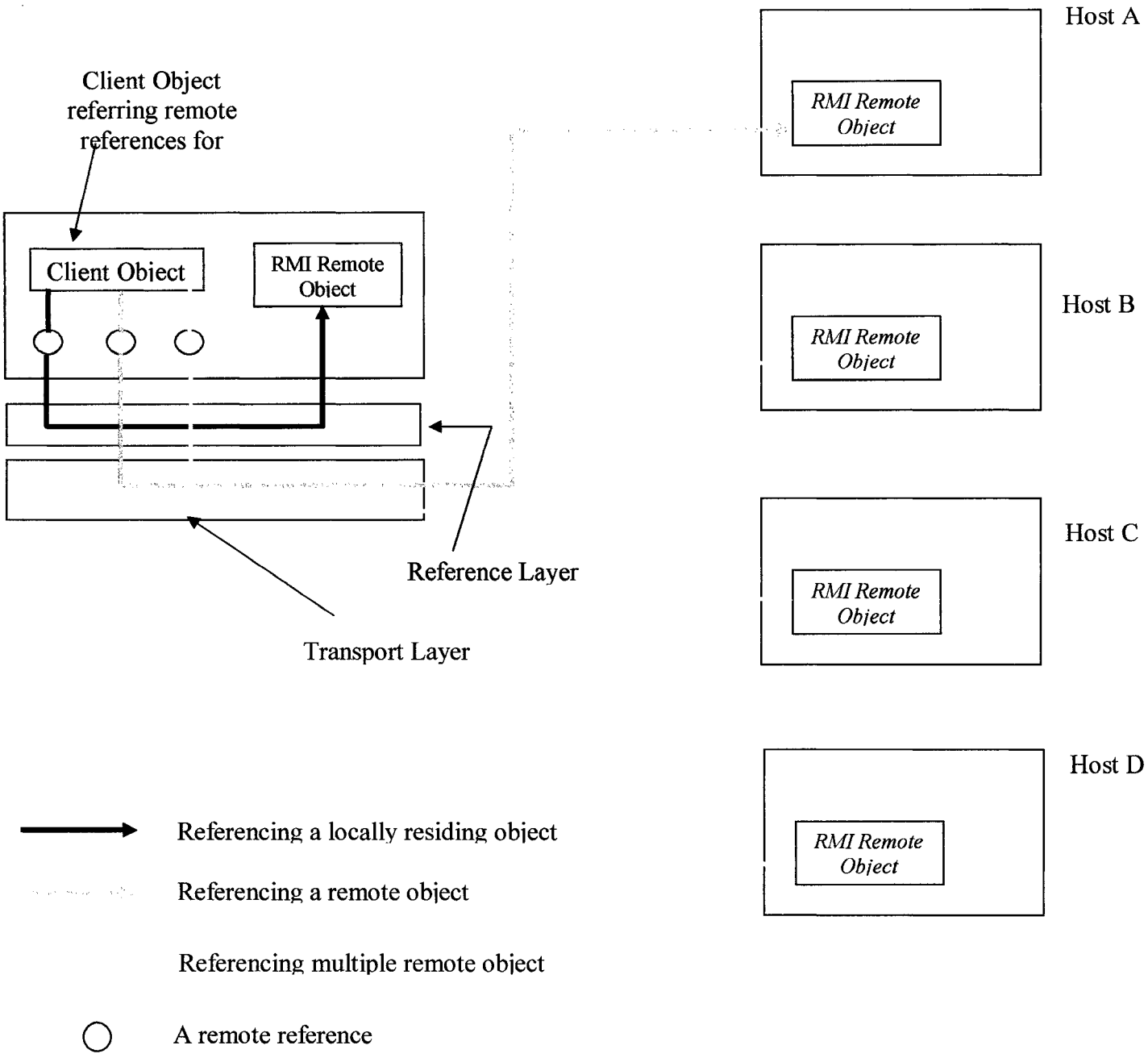


Figure 4.4. Multiple referencing semantics at Reference Layer

Passing Parameters/Return Values

The method call provided by the remote objects requires a transparent implementation of passing parameters and return values to/from remote methods. The current implementation uses the following approach for passing parameters.

- Primitive types (int, boolean, double, etc) are passed by value.

- Local object references are passed by copying the current object contents in the memory and serialize it to the remote location. Effectively, they are also passed by value. This is because the object references (memory addresses) are only valid in a local JVM.

  This clearly needs proper marshalling of all memory references relevant with the object. A reverse process is needed (un-marshalling) at the receiving end to bring a proper memory image of the received marshaled object. Figure 4.5 shows this process of parameter passing in a remote method invocation.

- Object references representing some remote objects are passed as remote references. As explained before, these references hold all the necessary details for contacting the remote object it refers to. The recipients can then invoke methods on the remote objects by making use of the remote references.

In order to hide the mentioned system level activity and to realize the traditional method call semantics (dot notation) RMI uses an additional layer of stub and skeleton objects.

- A stub object behaves as a surrogate for the remote object at the client location. It implements the same method signatures as used in the exported methods for the remote server object. Any invocation to a remote method by the client actually results in the invocation of the equivalent method signature in the stub object. However, the methods in the stub object only implements the logics for marshalling of method parameter values as received from the client. They also do un-marshalling when received marshaled return values from the remote (server) object.

- A skeleton object is a stub counterpart at the server site. It receives a method call request and the marshaled parameters, sent by the stub and communicates with the server object itself.

The code for stub and skeleton classes can directly be generated using a stub compiler, named ***rmic***, provided with the RMI package. A copy of stub byte code is then needed at the client site during a remote transaction. This can be done either statically (manually copying the stub code at client locations) or dynamically (using Java dynamic class loading feature).

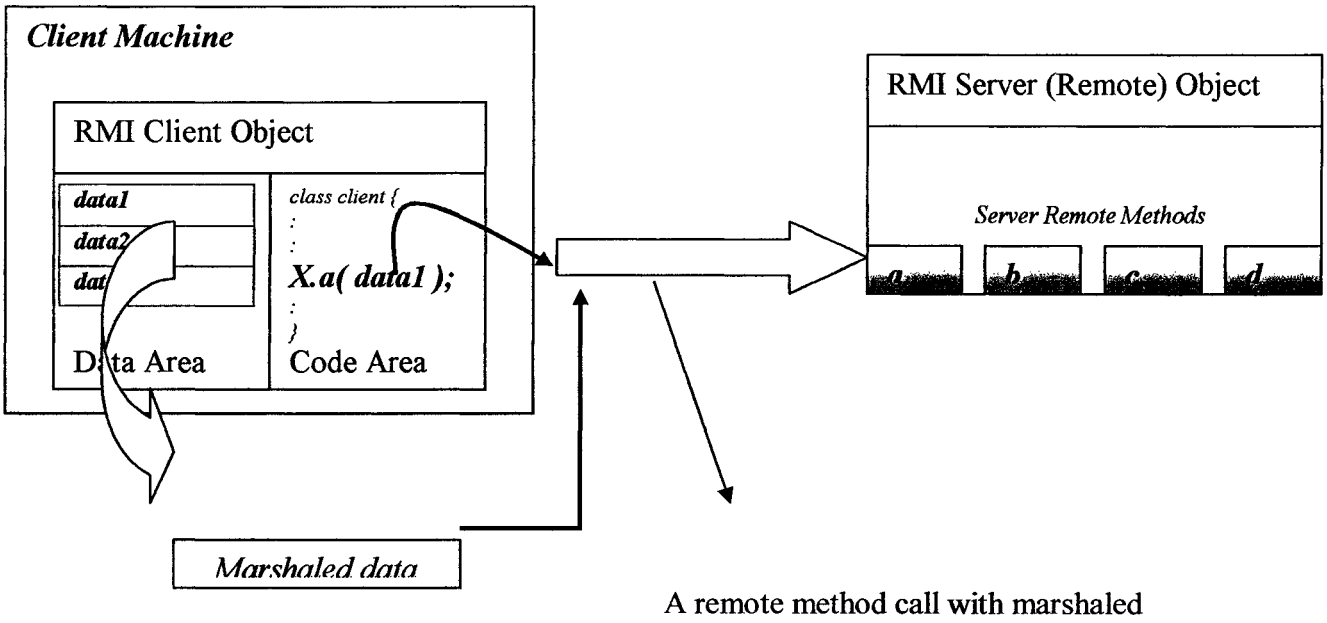


Figure 4.5. A Remote Method Invocation with Data Marshalling

4.2.3.2 Remote Transaction Scenario

Figure 4.6 illustrates a complete RMI working environment. In the scenario shown, a client object on machine xxx.yyy.zzz invokes a remote method in server object residing at aaa.bbb.ccc. A client call talks to a stub class. The stub passes the conversation along to the remote reference layer, which talks to the transport layer. The transport layer on the client passes the message across the network to the transport layer on the remote server. The server's transport layer then passes the message to the server's remote reference layer. As mentioned before, the main function of this layer is to build the desired semantic of this remote invocation i.e. this layer could invoke method on one or more than one server objects. In the other direction (server-to-client), this flow is simply reversed.

All the mentioned communication is implemented synchronously. That is, the client call continues waiting until the remote server's response is received.

4.2.3.3 RMI based Distributed Implementations

RMI provides the capability to write portable distributed objects. This is because the standardized Java virtual machine environment (JVM) on which RMI works, opens new opportunities to migrate and execute distributed code between heterogeneous hosts.
It means, these objects could be executed on any host irrespective of its architecture and could be used from anywhere. This provides the basis for a distributed programming system whose main purpose is to simultaneously execute multiple tasks in a distributed program using available network hosts.

A Generic Parallel Application Structure
Generally speaking, distributed application can consist of a number of tasks, each of which may be dynamically manifested as multiple instances that cooperate to solve a particular problem. Instances thus are independent sequential threads that may spawn or terminate other instances, synchronize with one or more, exchange data and do all that is needed to cooperatively solve the problem on hand.

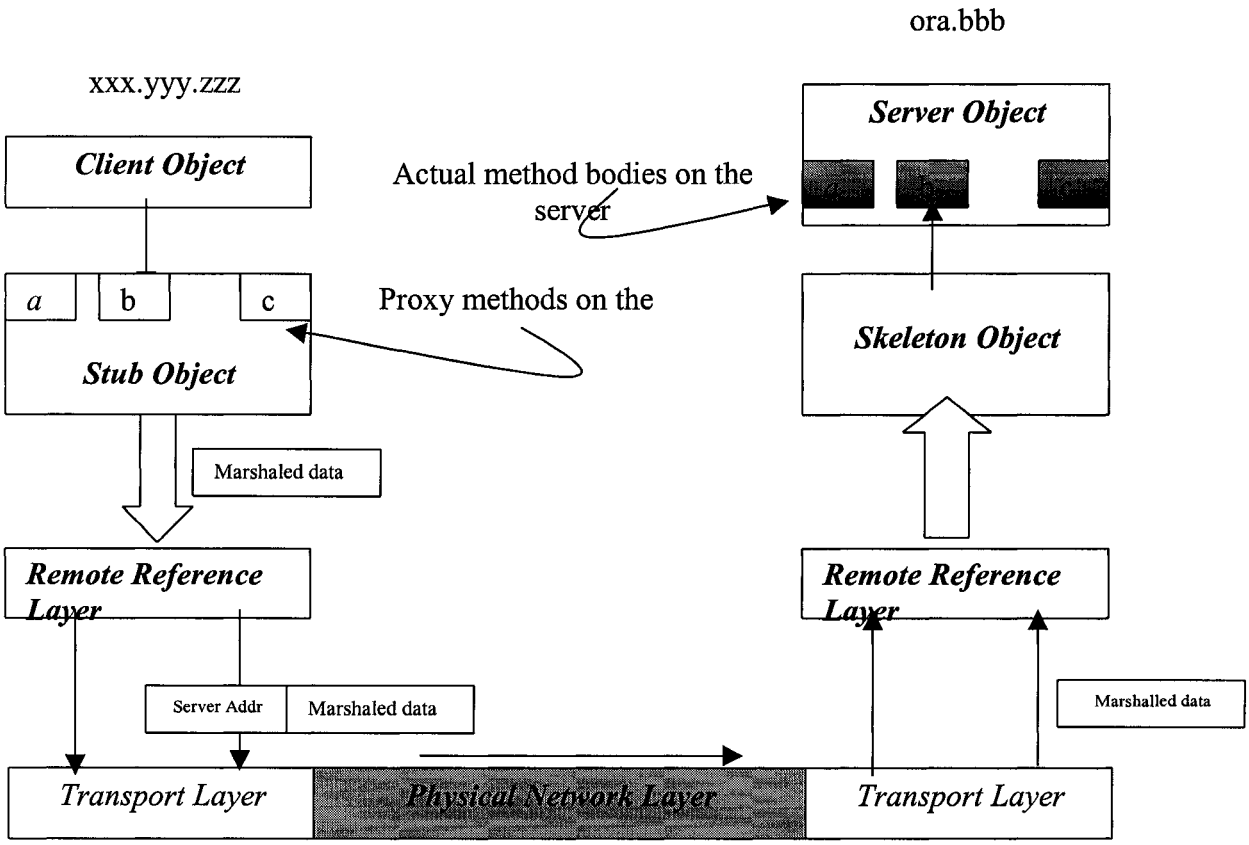


Figure 4.6. Communication of RMI Objects

Figure 4.7 shows a sample application having four components to solve some hypothetical problem. Each of the components represents a task instance. Further let ComponentA do the job of partitioning input and passing the suitable partitions over to the ComponentB. This takes its input and performs Matrix multiplication on it and returns the results to ComponentD. Similarly, ComponentC takes is set of partitions from ComponentA, performs Cholesky Factorization on it and returns the results to ComponentD.

Instances communicate via messages. Support for both blocking and non-blocking intercommunication is needed. Synchronization is based on barriers. This support is needed for cooperative processing based application where an instance after performing its execution must wait until the other processing elements catch up with it by sending appropriate messages.

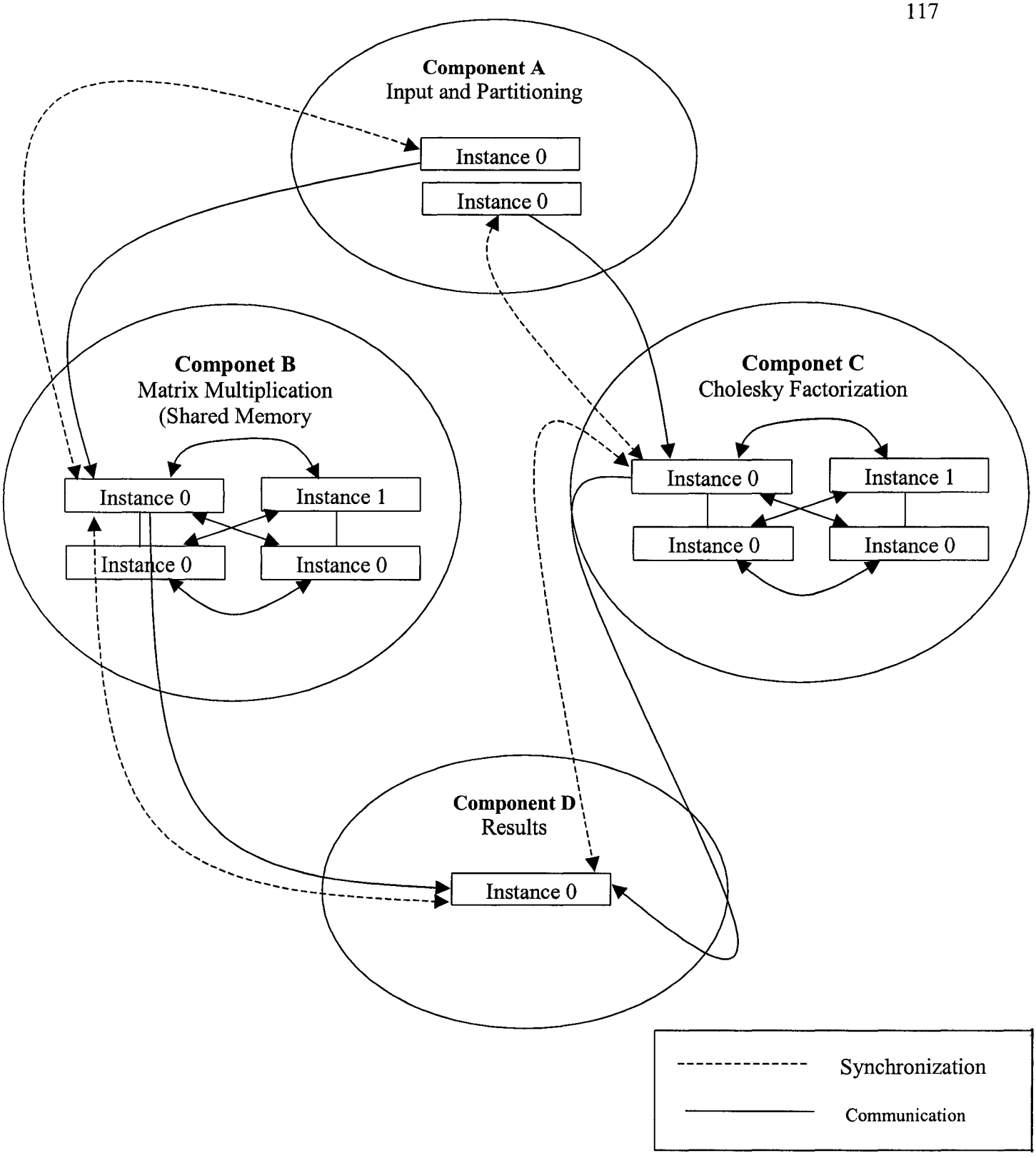


Figure 4.7. Example Structure of A Generic Parallel Application

4.2.3.4 RMI Support

The above-mentioned scenario of the parallel application can easily be implemented in an RMI environment, by realizing the task instances as remote objects. Each object implements computation logic for a task instance and provides method interface to be called by other objects. Remote references are then used throughout the system to refer to these objects remotely from any where in the system.

The RMI distributed model supports a robust inter task communication feature by providing traditional procedure call semantics with the existing remote objects. The tasks can send and receive messages to each other just by invoking the remote methods they support. This invocation also hides all the complexity of marshalling/unmarshalling for the data involved in this inter-task communication.

While running a parallel application, sometimes it is more beneficial to run some specific application task on some particular host. For example, in the above-mentioned application scenario, the matrix multiplication of ComponentB is more suitable for a shared memory multiprocessor. The number of instances depends upon the size of the matrix under consideration.

RMI environment supports this selection of the best-suited host for executing tasks of an application. However this requires the dynamic downloading of object's stub class code whenever it needs to be sent to some host. The current RMI implementation inherently supports this dynamic downloading feature and thus is easily adoptable for building such environments.

4.2.3.5 RMI Limitations

- One of the main limitations, which RMI imposes, is its synchronous communication model. This limits the tasks to communicate with each other in a synchronous manner thus imposing unnecessary delays, which could be avoided if

asynchronous method calls could be supported. However the future releases of RMI are supposed to overcome this limitation.

- Another limitation is the limited object registration support. Current implementation allows registration of remote objects created on the local host only. This limits the possibility of having a global registration host for all remote objects in the environment. Implementing the additional logics for dynamic downloading of stub code could provide a global remote registration support.

  JLBS extends the object registration feature by allowing the registration of remote objects at a central registry host. This requires the availability of object's stub class at the central registry host. JLBS dynamically loads stub byte codes during a remote registration process.

  Figure 4.8. shows a complete scenario for a remote registration activity. A central registry mechanism is working at the host, named regHost.ora.com. Another host xxx.ora.com is having a remote object named X. The figure demonstrates the process of registering the object X at the central host regHost.ora.com. Two additional objects are used to implement the dynamic downloading of stub code: A socket based object at host xxx.ora.com and a class loader object at host regHost.ora.com.

  The process starts with a registration request came at a well known registry port for object X, initiated from host xxx.ora.com. For having the object registered at the host regHost.ora.com, its stub code is requested. The class loader object at host regHost.ora.com forward this request to the socket based daemon at host xxx.ora.com. The daemon reads the stub code from the local file system and forwards it back to the class loader. This completes the registration process.

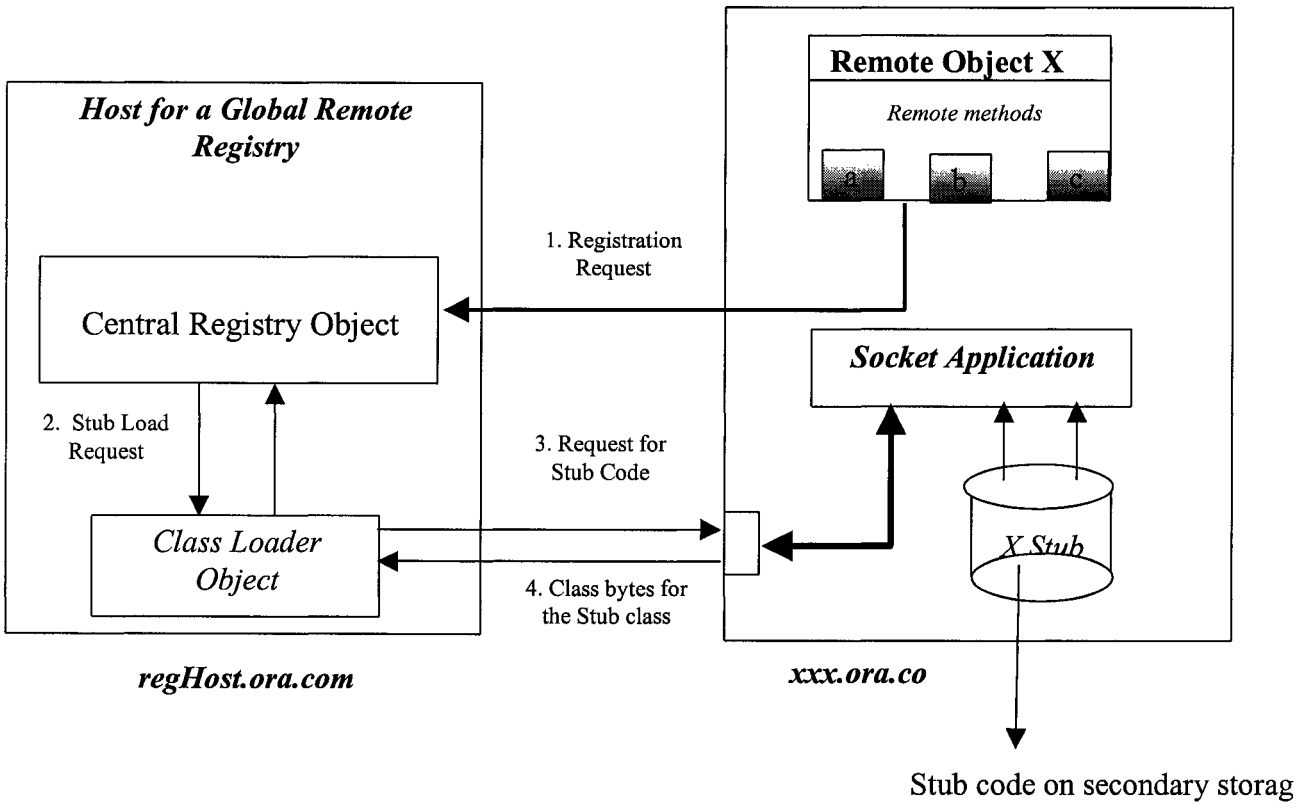


Figure 4.8. Scenario Showing A Remote Registration Process

4.3 Details of JLBS User Development Support

As explained in the previous sections, JLBS provides a class library interface for developing parallel and distributed applications. As explained in section 3.8, every parallel application is written as a collection of one application agent and a number of of application task objects.

The application agent mainly performs control activities relevant with the usage of system while task objects mainly perform the application on hand. In addition to this, there exist interactions between application agent and application task objects, which mainly account for the partitioning of application data and collection of results.

This section will explain the steps needed to develop an example parallel application using JLBS development libraries. Mainly three steps are needed for any such development.

1. Defining formats for task input and output data sets
2. Defining an application agent code
3. Defining application task object code

4.3.1 Defining formats for task input/output data sets

In any parallel application based on data partitioning, each application task will receive a partition of application data and work on it in parallel. Depending upon the paradigm used, each task then communicates its results to other tasks in the application and all the generated results will then be integrated into a final collective application output.

Thus while writing tasks for a parallel application, the formats for their input and output data sets should be clearly defined. The same definitions will then be followed in the tasks computation codes.

JLBS requires the developers to define formats for the input and output task data sets.

Two class interfaces, one for each input and output record formats, are provided in the package named ***jlbs.development.base.*** They are named as ***TaskInputRecord*** and ***TaskOutputRecord*** respectively.

4.3.2 Defining An Application Agent Code

Developers need to use a JLBS library class, named ***JlbsAppObject,*** for extending their application specific application agent class. The parent class is already provided with the the code concerning the main activities of application agents. As defined previously, the main activities performed by the application agent class object are as follows:

- The object needs to communicate with the JLBS for the starting and ending a parallel application run and for initiating its individual task executions.
- The object contains the complete input data set for the application, which it needs to distribute among the running application tasks.

In the extended class, the developers need to write

- the ***run*** method which is responsible to integrate the above mentioned activities in an application specific manner, and
- the receiveTaskResult method, which accounts for receiving the individual task result records and integrate them in the final application output.

The method names in the parent class, their descriptions and the sequence of their invocations are discussed in Table 4.2.

JlbsAppObject Class	
Description The class is used for implementing the application agent object code. It contains methods which implements the logics for most of the activities needed by an application agent object. The developers need to invoke these already built methods with the parameters specific to their application code.	
Method Interface used by the application agent code extension	
Method Name	*Description*
StartApplication	Method for a logical startup of a parallel application. The method registers the parallel application on hand by initiating the relevant memory data structures of the JLBS system with application specific values. Arguments: None
EndApplication	The method brings a logical end to a running parallel application. These may involve logging of certain application specific data, obtained from the recent run, on permanent system logs. Arguments: None
ExecuteTask	The method for parallel execution of a specified number of application's task on the currently available system hosts. Prior to its execution, an automatic downloading of task executable code is performed from its source to the remote host. ***Arguments***: application name, task name, package name, number of task instances needed.
WaitForResults	Basically it allows the application agent code to wait for results from a specified number of application tasks until it could do further processing. ***Arguments***: Number of tasks need to be looked for during the wait period.
SendTaskDataSet	Allows dispatching a data input partition to the specified application task. The format of the data being supplied to the task

	should already agreed upon and well understood by the receiving entity. ***Arguments***: application name, task name, package name, the task instance (in case if multiple instances are running), the task input data record.
SetupTaskLinks	Allows to create communication links between two specified tasks, the source and destination tasks. The nature of links created between the specified tasks is bi-directional i.e. any of the tasks involved could use it in any direction. These links will then be used to send and receive application specific data during an application run. ***Arguments***: name of two tasks involve in communications.
ParallelTaskTriggering	Notifies an application task for starting its computation loop. The method is invoked when the task being addressed has already received its complete data set and the task communication links are properly setup. ***Arguments***: application name, task name, package name, task instance (in case if multiple instances are running)
Method Interface used by the task object	
CollectChildRef	Used to submit a communication link of an application task, just spawned. This will allow the application agent to access the system specific methods of the running task objects by making use of the link received. ***Arguments:*** application name, task name, task remote reference
ReceiveTaskResultSet	Used by the task object to send back the task results to the application agent. The format of the data being received should already be agreed upon and well understood by the application agent code. ***Arguments***: An standardized task output data record.

Table 4.2 Method Interface for JlbsAppObject Class

4.3.3 Defining Application Task Object Code

For writing an application task object, JLBS user library provides a class, named ***JlbsTaskObject.*** The class resides in the package named ***jlbs.development.base*** and provides many system level activities transparently.
The activities a task object to provide are:

- communicating with the application agent for receiving the task's input data set.
- communicating with the application agent for supplying the task's output data set.
- conducting the actual application specific computation on the task input data set.

The routines for realizing application agent communications are already provided with the parent object in the JLBS development library. The developers are supposed to provide two method bodies in their class extensions. They are as follows:

- A method, named ***receivedTaskData***, for retrieving records for task input data and then properly extract the data set from those.
- A method, named ***compute***, for application specific computation over the task's input data set. In case of multiple instances of a task object, the same compute code will work with different partitions of data supplied by the central data distributor (application agent).

The compute method should not start the computation until the complete input data set is retrieved. The JLBS environment ensures this by invoking the compute method only after receiving an *"input data received notification"* from the application agent object. At the end of its execution, the compute method builds a task output data record and send it to the application agent object.

The details of method interface provided in the parent class for application task objects are specified in Table 4.3.

JlbsTaskObject Class	
Description	
The class is used for implementing the application specific logic of task computations in a parallel application. It contains already built methods which implement the logics for most of the system based activities need to be done in an application task.	
Method Interface used by the task object code extension	
Method Name	*Description*
SendTaskResultSet	Used at the end of the task computation method in order to send a task output data set to the specified task in the running application. *Argument*: TaskOutputRecord (the data structure format is different for different applications).
Method Interface invoked by the application agent code	
ReceiveTaskData	Invoked by the *SendTaskDataSet* method of *application agent* object and used for submitting the task input data records. The format of the data being received should already be agreed upon and well understood by the application agent code. ***Arguments***: A standardized task input data record.
InvokeTaskRun	Invoked by the TaskTriggering methods of the application agent object. It causes the task compute method to be started immediately. ***Arguments***: None
AddToTaskInputList	Invoked by the *SetupTaskLink* method of the application object class. It results in building a communication link between the task addressed to some other task specified in the method parameter. ***Arguments***: Task identification record for the source task in the link
AddToTaskOutputList	Invoked by the *SetupTaskLink* method of the application object class. It results in building a communication link between the task addressed to some other task specified in the method parameter. ***Arguments***: Task identification record for the destination task in the link

Table 4.3 Method Interface for JlbsTaskObject Class

4.4 Writing an Example Application – The Dot Product

4.4.1 Problem Definition

The simple example chosen is the dot product problem of two vectors. "Given two vectors ***a*** and ***b*** of n dimensions, the dot product or inner product of these two vectors is given by the sum of product relation.

The above problem could be solved using a supervisor worker paradigm by having ***n*** clients or workers computing sum of products on ***n*** different portions of both vectors. This will result in ***n*** partial values of the complete dot product. All these values will then be collected by a central supervisor and are then added together to find the final value for the complete scalar product. Figure 4.9 shows a pictorial view of this parallel application structure.

4.4.2 Implementation

The above algorithm will be implemented using the following steps.

- The formats for task input and output data records are defined.
- A supervisor task is created by extending from JlbsAppObject class and writing the run method.
- A worker task class is created. The n copies of the same code will then be used by the supervisor object to solve the problem.

4.4.2.1 Defining Task Data Formats

The complete application input data sets consists of two large vectors (linear arrays). The supervisor task (application agent) is to spawn n instances of client code (application task objects), decompose the complete data set into n distinct instances and send them to the created application tasks.

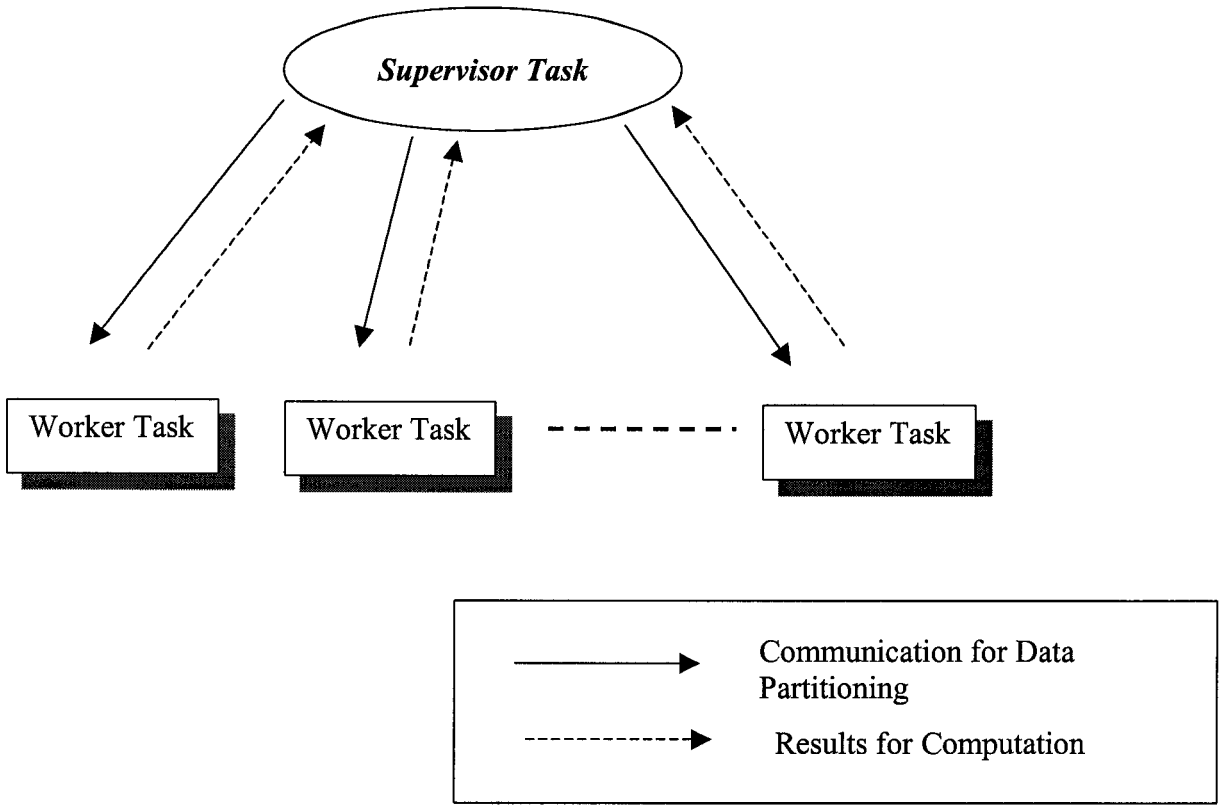


Figure 4.9. Scenario showing a parallel computation of a dot product of two vectors

Each task input could be thought off as two sub-arrays with an identification numeric tag for this computation instance. This tag value could then be used in proper sequencing the received results in the supervisor task. However due to the commutative nature of dot product, this tagging is optional here. Figure 4.10 shows the format used for task inputs for this example.

The result of each of such computation is a scalar value. Thus the output record should have a field for this value plus any additional control data. Figure 4.11 shows the output record format for each task.

```
Task Input Data Format {
   1. Numeric Computation Tag
   2. Sub-array for first vector
   3. Sub-array for second vector
}
```

Figure 4.10. Task Input Data Format for worker's task

```
Task output data format
{
   1. Numeric Computation Tag
   2. Scalar product (partial) value obtained
      from dot product computation from each
      worker.
```

Figure 4.11. Task Output Data Format for worker's task

4.4.2.2 Writing A Supervisor Object Code

The supervisor class is extended from the main library class, JlbsAppObject. The application writer then codes the run method for the extended object. The run method code must go for a proper calling sequence of the parent object system routines. For example, in our case the following sequence will be adopted.

MethodName: run

Input: None

Output: n^1 copies of worker tasks are instantiated with separate data partitions and their results are then collected back.

Processing:

- Register the parallel application with the JLBS environment
- Decompose the problem in *n* partitions.
- Execute all n copies of worker code.
- Decompose the application data into sub-partitions for distributing them to the running tasks.
- For (all n tasks running in the system)
    - Send different data partitions to its respective task running code.
    - Trigger the task compute method
- EndFor
- Wait for the n tasks for returning of their computation results.
- If wait ends, collect the result and integrate them to some application specific manner
- Return the final result.

EndProcessing

4.4.2.3 Writing A Worker Object Code

For the worker class, the library class JlbsTaskObject will be used. The application writer needs to write two main methods for its code extension: the compute method and the receiveTaskData method. The code for receiveTaskData receives the task input data records while the compute method implements the actual computation on the received data. The sample code for both methods are listed below:

MethodName: receiveTaskData

Input: TaskInputDataRecord

Output: Task data area has been filled with the values supplied in the task input record.

Processing:

Receive a reference to task input data record as a parameter.

Extract

- the task subarrays (data partitioning) needed to compute the partial dot product.
- the task computation identification tag.

EndProcessing

MethodName: compute

Input: None

Output: The computed value of partial dot product.

Processing:

Read arrays from task data area to find the dot product.

Compute the dot product as a partial sum of products formula.

Using the value computed and the task computing identification tag, create a task output data record.

Send the task output data record to the supervisor task (application agent).

EndProcessing

4.5 JLBS Algorithm Details

This section describes main JLBS algorithms. These can be categorized into the following categories.

- Algorithms related with an application execution
- Algorithms related with task spawn map generations
- Algorithms related with monitoring a running task.
- Algorithms related with application history maintenance
- Algorithms related with load collection

4.5.1 Algorithms related with an application execution

These algorithms implement the logics related with the application executions. Each of them is implemented by invoking methods from different system objects.

4.5.1.1 Setup an application running environment

The algorithm builds an application runtime environment at the startup of a parallel application, which is needed for a load-balanced run. The algorithm's main purpose is to load the application history data in memory. This information will be used at runtime for load balancing decisions. The details are listed in Algorithm 4.5.1.1.

The algorithm is mainly implemented by the setupApplicationRunTime() method of *JlbsHistoryManager* object. The algorithm is triggered by the start() method of *JlbsApplicationObject* used for starting a user application run. This method then invokes startApplication() method of *JlbsMainController* object which in turn calls the setupApplicationRunTime() method in *JlbsHistoryManager* object.

The algorithm takes as input, the name of the application about to run and the host name from which it is arriving. At first the algorithm checks whether an application with the same name is already running. If yes, the system denies any further processing, as it doesn't support more than one run of an application at the same time. The main reason is

that the task monitoring records received by the system for a running application with multiple copies can not be distinguished for different copies of the run.

If there is no running copy of the application, a new structure is allocated for running the application in memory. Contacting the application's source host then populates the fields for that structure. These include the number of tasks in the application and the sum of file sizes of its executables. A unique application name is then computed from them and the history repository is then searched for this name. If the application's name is found, it means that this application has run before and its history data can be used in load balancing decisions. In case the search failed, the application is marked as running for the first time and default values are assumed for the application's first run.

4.5.1.2 Spawning an application task

The algorithm results in instantiating an application task on a remote host. The main purpose is to implement the physical task dispatching on remote hosts. The details can be seen in 4.5.1.2.

The algorithm is implemented collectively by the methods of application agent, main controller and task agent objects. The application agent's executeTask() method is invoked whenever there is a need to remotely spawn an application's task. The parameters needed by the method are the task name, its application and package names, the task source host name, number of copies needs to be spawned and a communication link reference for the application agent. The method forwards the request to the main controller object in its own domain.

On the basis of the current spawn method specified in the system configuration file[2], main controller generates a task spawn list, which tells about the task spawn details on different system hosts. It then contacts the task agent objects on each of the hosts included in the task spawn list with a task execution request. The task agent objects then contacts the daemon object on the task originating host and download the actual task executable from

there. After that each task agent instantiated the task copies specified in the request and starts monitoring their execution.

Algorithm 4.5.1.1: Setup an application running environment

Input: Name of the application, Name of the application source host

Result: An application history block with its history data loaded in memory.

Processing:

Check whether the application is already running.

If (not running) Then {

    Allocate a new application structure in the current history manager object.

    Build a communication link with the application source host to download the application's specific parameters.

    Compute a unique application name used for searching the history repository.

    Search for the application's existence in the history repository table on the domain control host.

    If (application found in history repository) Then

        Load application history data from the history repository

    Else

        Initialize application in-memory structure with default values

    EndIf

}

Else

    Give error message: Cannot run an application which is already running

EndIf

End Processing

Algorithm 4.5.1.1 Setup an application running environment

[2] Name of the configuration file is jlbs.conf and is placed in the application running folder.

Algorithm 4.5.1.2: Spawning task of an application

Input: Task identification (package/application/task names), name of the host originating the request, number of instances needed, a communication link reference to the task's application agent.

Result: Instantiation of the task copies on different remote hosts

Processing:

Application agent generates execute task request for the its domain main controller object.

Using the task history characteristics and the current spawn method (balanced/fixed/random) specified in the system configuration file, MainController object generates a task spawn ma for the requested task.

For (each host in the task spawn list) Do

Invoke the executeTask() methods of the host task agent object, for executing the specified number of tasks.

EndFor

End Processing

Algorithm 4.5.1.2 Spawning task of an application

4.5.1.3 Ending a running application

The algorithm mainly implements the logic for ending an application run. It results in updating the application history records and the de-allocation of resources used for the application run time environment.

The algorithm is collectively implemented by the methods in application agent, main controller, task agent and history manager objects. The details of the algorithms are as in Algorithm 4.5.1.3.

Algorithm 4.5.1.3: Ending a running application
Input: User perceived application name

Result: Application history records are updated in the history repository and the resources in use are de-allocated.

Processing:

The application agent generates an end application request to the main controller object of its domain.
Main controller generates a request to its local history manager object for updating the application history record.
The application history record is then retrieved.
For (each task included in the application) Do
    Retrieve the task spawn map, generated at the time of task spawn.
    For (each host in the task spawn list) Do
        Task recent execution behavior is requested.
        Task history record is then updated with the recent execution behavior.
    EndFor
EndFor
The updated application history is then stored in the history repository for the local domain.
For (each existing domains) Do
    Main controllers are requested to update their local history repositories with the current application run.
EndFor
Remove the application history structure from the runtime history manager object.
End Processing

Algorithm 4.5.1.3 Ending a running application

4.5.2 Algorithms related with task spawn map generations

The algorithms implement the actual logic for generating the task runtime spawn lists. These lists specify the task spawn details on the basis of criteria specified in the system configuration files. In the current implementation, the generation is based on three main criteria including balanced, fixed and random generation. A single class, named JlbsLoadBalancer, implements all such algorithms.

4.5.2.1 Load Balanced Task Spawn Algorithms

Selecting a domain for a load balanced task spawn

The basic idea behind the load balancing problem is to efficiently utilize the system resources by making use of the least loaded hosts for task executions. However JLBS adds one more abstraction level of host domains.

Each domain load is represented by a load index value, which is an average of the current load indices of all hosts included in that domain. While performing a load balancing execution, the system first checks whether the local domain is suited for initiating a load balanced run. Comparing the current domain load index with the domain load threshold value, assigned to it by the domain configuration file, allows checking for this possibility.

In case of a heavy loaded domain, the algorithm then goes for selecting a least loaded domain from the existing ones. It takes as input a list of all domain main controller objects, which will be used to compute their current domain load indices. The selected domain is then used for initiating a load balanced run for the requested application. Algorithm 4.5.2.1.1 shows the steps involved in this selection.

Algorithm 4.5.2.1.1 Selecting a domain for load balanced task spawn

Input: A list of main controller objects for all existing domains.

Output: A main controller object reference for the most lightly loaded domain

Processing:

Compare the current domain load index with the load threshold value (from the configuration file).

If (local domain load index is less than the threshold value) Then

    Generate task spawn map for the local domain.

Else {

    Select the domain with the least load index

    Generate task spawn map for the selected domain

}

EndProcessing

Algorithm 4.5.2.1.1 Selecting a domain for load balanced task spawn

Generating a load balanced task spawn map

The algorithm actually implements the load balancing strategy for JLBS. It does a balanced task execution within the boundaries of a selected domain. It takes as input the task name, the number of task instances to be spawned, list of domain host loads, list of task dispatcher (task agents objects) links on each domain host and the task history record. The detailed algorithm can be seen in Algorithm 4.5.2.1.2.

The algorithm tests the feasibility of running the requested task on each host of the selected domain. It first computes the task execution time (predicted value) and the volume of communication in bytes between the said task and all the assigned ones so far. It then converts this value in bytes into time in seconds based on certain characteristics of the host being considered (Algorithm 4.5.2.1.3) . On the basis of these values, a task cost is then computed for each domain host.

Now, for each host the number of task copies to be assigned is computed. The hosts where the task running costs are comparatively lower are naturally good candidates for the running more task copies.

Algorithm 4.5.2.1.2: Generating a load balanced task spawn map
Input: Application name, task name, package name, number of instances, list of domain task agent objects, list of host load records for the domain, task history record
Output: A load balanced mapping of the task to the domain hosts.
Processing:
For (each host in the domain) Do
    Evaluate the cost of running the task under consideration on that host (Algorithm 4.1.2.4)
    Save the computed cost for the task run in a cost array
EndFor
Sort the cost array in increasing order
Calculate the ratio of distribution of task copies to the domain hosts.
Generate the task spawn list for the domain hosts. Each record in the list has the following format.
    < Host name, Task instances, Object reference for host's task agent >
Return the spawn list.
End Possessing

Algorithm 4.5.2.1.2 Generating a load balanced task spawn map

Compute host characteristics parameter
The algorithm implements a heuristic to compute a host characteristic value by using certain host data. These data are supplied by means of host specification file on each host. The selection for host data is not standardized. In order to precisely denote a host capability, fairly complex heuristics could be used. However, simple heuristics using less computation power, are proved to be satisfactory in most situations.
This implementation uses a simple weighted average formula to compute the host characteristic value. The following weights are given to the specified host characteristics.

- 60% to the collective MIS of the processor,
- 25% to the main memory size and
- 15% to the swap space available.

This computation will be the same in all load balancing heuristic experimented.

Algorithm 4.5.2.1.3: Compute host characteristic parameters

Input: Name of the host characteristic file

Output: A single integer value to represent the host capability

Processing:

Read host specification file parameters.

Use these parameters to compute host speed index for representing the host processing power.

Return the computed value.

EndProcessing

Algoithm 4.5.2.1.3 Compute host characteristics parameter

Algorithm 4.5.2.1.4: Compute a task execution cost for a given host

Input: History record for the task to be executed, target host name

Output: Predicted cost if the task is executed on the given host

Processing

Let s be a measure of power of the host (host speed/memory etc.)

Let l be the normalized load on the host at that instant

Let x be the weighted average of completion times of the task with respect to the number of runs

Let c be the communication cost between all the currently executing tasks on this host with which the subject task communicates

Predicted Cost = l * s * x + c

End Processing

Algorithm 4.5.2.1.4 Compute a task execution cost for a given host

The generated spawn list is then processed by the main controller object to do the job of actual task dispatching to the selected hosts in the list.

4.5.2.2 A Random Task Spawn Map Generation

The heuristic tries to run a parallel application by randomly assigning its task to different hosts in the virtual machine (Algorithm 4.5.2.2). The approach is proved to produce satisfactory balancing environments in some situations, especially in near to uniform dynamic load on hosts. It does not take into account any task, application or system information in account for the purpose of task distribution. For each request to execute a task, all it does is to generate a random number between 0 and N – 1 where N is the number of hosts in the current domain. The task is then sent to the selected host. The details are in Algorithm 4.5.2.2.

4.5.2.3 A Round Robin Task Spawn

This heuristic also does not take any application information into consideration. The algorithm uses equal distribution of application load between the available hosts in all domains. The strategy applied for implementing this is that an instance of each task execution request is given to the available hosts in cyclic order. Thus if m task execution requests arrive, and m = k*n where n is the number of hosts in a domain and k is a constant, then every domain host gets k task instances to execute, as in Algorithm 4.5.2.3.

Algorithm 4.5.2.2: A Random Task Spawn map generation
Input: Application name, task name, package name, number of instances to spawn, a list of domain main controller objects

Output: A randomly generated mapping of the task to the system hosts
Processing:

```
For (each domain in the environment) {
        Get the domain host array for the domain
                Select any host randomly.
                For the current task, randomly select the number of copies to be assigned to the
                selected host.
                If (task copies assigned >0) {
                        Add the domain host in the task spawn list. Each record in the list has
                        the following format.
                        <Host name, Required instances, host task agent  reference>
                }
                If (total copies assigned is equal to the total instances need to be assigned)
                        Returns the computed spawn list
        EndFor
EndFor
```

End Possessing

Algorithm 4.5.2.2 A Random Task Spawn map generation

Algorithm 4.5.2.3: A Round Robin map generation
Input: Application name, task name, package name, number of instances to spawn, a list of domai main controller objects

Output: A mapping of the task to the system hosts generated in a round robin fashion
Processing:

```
For (each main controller in the domain list) {
        Get the domain host array for that domain
        For (each host in the domain)
                Assign one task copy to be run on the host
                Add the domain host in the task spawn list. Each record in the list has the followin
                format.
                < Host name, Task instances, Object reference for host's task agent >
                If (total copies assigned is equal to the total instances need to be assigned)
                        Returns the computed spawn list
        EndFor
EndFor
```

End Possessing

Algorithm 4.5.2.3 A Round Robin map generation

4.5.3 Algorithms related with task monitoring

Each task executed in the JLBS environment, needs to be monitored to record its recent execution behavior. The monitoring is actually performed by the low-level system routines. The task agent objects gather and log this data on each host by using a local task execution manager object. The object provides services to easily maintain and access this task characteristic data log.

The monitoring data captured for each task consists of the execution time taken by the task to complete and the task communication behavior. The memory layout of the data structures used by the task execution manager object is shown in Figure 4.12. The following can be noted from the figure.

- Each task is assigned a unique identification string, called the task id. The string will be used to uniquely address the task in the object.
- Each task node points to a linked list of tasks that it communicates with. Each node in the task list gives details of communication volume.

Algorithms listed in the coming sections implement the whole logic of running task monitoring.

4.5.3.1 Initiate monitoring infrastructure

The algorithm results in having a setup for gathering local monitoring data on each host. It is implemented in the startup code for task agent object running locally on each host. Each task agent object spawns a background thread for monitoring data collection and links it with the low level system routines performing actual monitoring. It then constantly polls the thread for logging any available monitoring. The details can be seen in Algorithm 4.5.3.1.

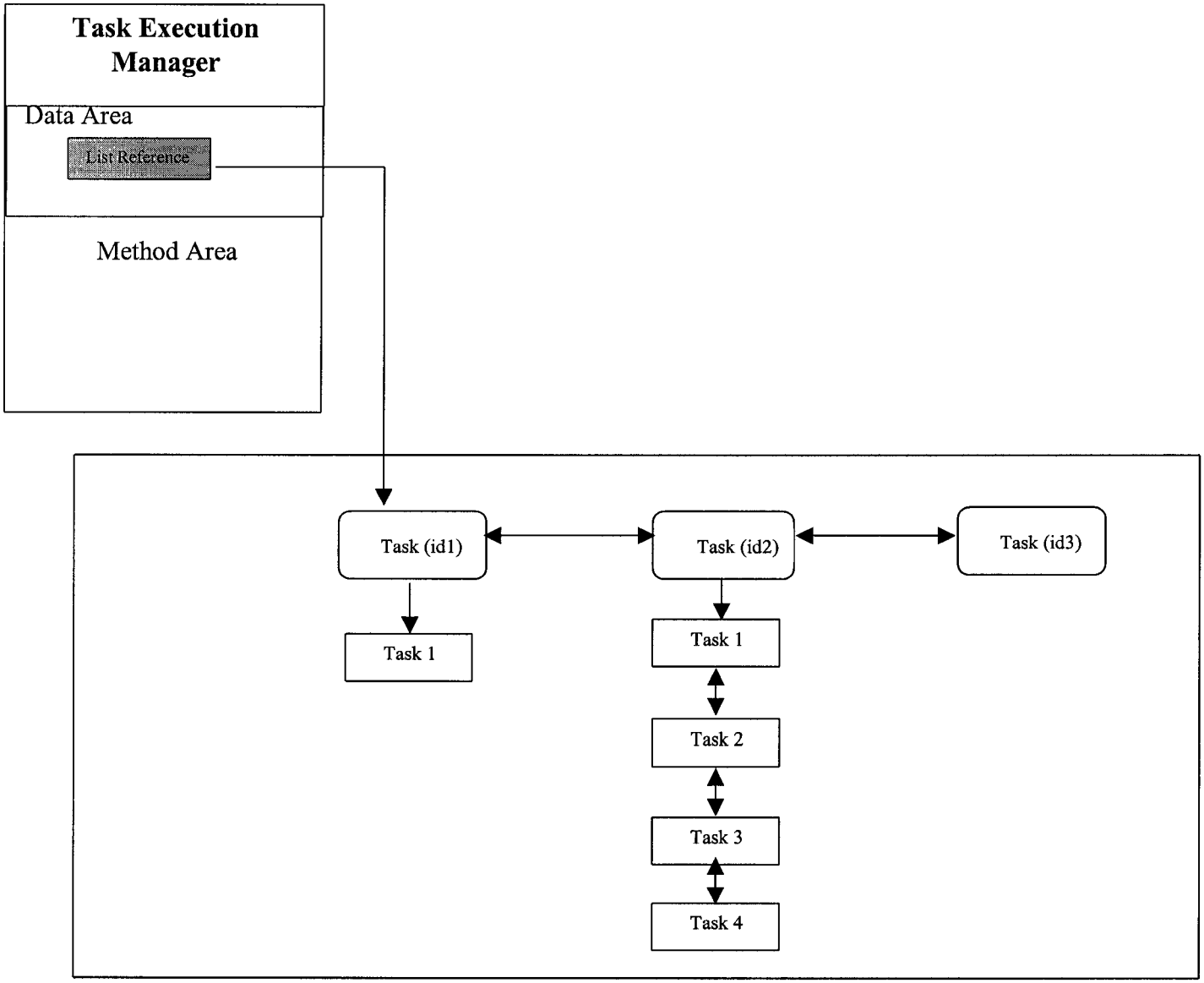


Figure 4.12. A Memory Image Of Task Execution Manager Object

Algorithm 4.5.3.1: Initiate monitoring infrastructure

Input: None

Output: A thread for reading task monitoring statistics is activated in the background.

Processing:

Initiate a background thread for receiving the task monitoring data from the underlying system routines.

Establish a communication pipe with the background thread.

Run an infinite loop which does the following:

Regularly poll the communication pipe for any data from the monitoring thread.

If (found any monitoring data) Then {

Received the monitoring data[3].

Logs the data in the in-memory task monitoring database[4].

}

End Processing

Algorithm 4.5.3.1 Initiate monitoring infrastructure

[3] It consists of the communicating task name and the amount of data it communicates.

[4] A unique task id string received with the monitoring data identifies the task for which data is logged.

4.5.3.2 Execute/Monitor a given task

The algorithm is implemented by the execute task method of the task agent object. It takes task identification, the host, which has a task executable code and the number of task copies need to be spawned. For each task copy to be spawned, a unique identification string is first obtained. This string will then be used in the task execution manager object to uniquely address the task while monitoring. The method then dispatch the task from its local host, instantiated the desired number of copies and starts logging its monitoring data received from the underlying system routines. The algorithm steps are listed in Algorithm 4.5.3.2.

Algorithm 4.5.3.2 : Execute/Monitor a given task

Input: task identification parameters (name of package/application/task), name of task originatin host, number of instances to be spawned

Output: The requested task copies are instantiated with the system routines start monitoring it.

Processing:

While (task copies spawned is less than the requested task copies) Do

From the task-originating host, load the executable class code.

Generate a unique identification string for the task to be spawned.

Spawn the task copy with the following task parameters supplied.

- the unique id task string generated for the task.
- the communication link for the task parent.
- link for the task agent object.

Refresh the domain load information after the task run.

Initiate a task monitoring structure in task execution manager object, for logging the tas running characteristics.

Use the unique task identification string to uniquely address the task in the task-monitoring log.

EndWhile

Algorithm 4.5.3.2 Execute/Monitor a given task

4.5.3.3 Update task execution time

The algorithm (Algorithm 4.5.3.3) updates the task monitoring structure with the recent task characteristics. The routine is implemented by the setTaskEndTime() method of task execution manager object. When a task finishes its run, it calls its local task agent object with its identification string. Task agent then logs the task execution time.

Algorithm 4.5.3.3: Update task execution time

Input: Task unique id string

Output: Updated task monitoring structure

Processing:

Receive the task-finished message from the finishing task.

Get the time of message receipt.

Search for the task monitoring record inside the task monitoring database.

Get the start time value for the searched task.

Compute the task completion time

Update the task completion time field in the monitoring record.

EndProcessing

Algorithm 4.5.3.3 Update task execution time

4.5.3.4 Update task communication behavior

This algorithm (Algorithm 4.5.3.4) logs a the task communication behavior. The algorithm is implemented by the updateCommData() method of the task monitoring object. The method takes the task identification string, the communicating task name, and the amount of communication in bytes.

Algorithm 4.5.3.4 : Update task communication behavior

Input: Unique id string for the task, name of the communicating task, number of bytes communicated

Output: Task communication records are updated in the monitoring data base.

Processing:

Find the task monitoring record from the task execution data base.

If (the communicating task name is already in the monitoring record) Then

    Add up the byte count to the already received one

Else

    Create a node for the communicating task

    Initialize the byte count to the value reported.

EndIf

EndProcessing

Algorithm 4.5.3.4 Update task communication behavior

4.5.3.5 Communicate task monitoring data

The domain main controller calls this routine at the time of ending an application run. For every application task, its recent task monitoring data is needed for updating the application history repository. Main controller dictates task agent objects at each host to extract and return the task recent monitoring data from its local task monitoring object.

The routine takes the application name and task name as input to identify the task in the monitoring database. The detailed steps can be seen in Algorithm 4.5.3.5.

Algorithm 4.5.3.5 : Communicate task monitoring data

Input: Application name, task name

Output: Record containing task monitoring data

Processing:

Get the list of the task monitoring records for all the instances of the running task on the local host.

For (each task record in the task list) Do

Get the unique task id string

Get the start time and end time values for the task id.

Compute the task completion time value.

Sum up the completion time value.

Get the list of the communicating tasks.

EndFor

Compute the average of the task completion time value.

Create a task monitoring record with the following format.

<Application Name, Task name,. Completion time average, Communicating task list>

Return the task monitoring record

EndProcessing

Algorithm 4.5.3.5 Communicate task monitoring data

4.5.4 Algorithms related to application history maintenance

In order to have an estimate of a task execution behavior, a history log of its running characteristics is maintained. This history log needs to be replicated in each existing domain. The main controller object makes use of a local history manager object to perform the main job of application history maintenance.

The history manager object contains methods and data structures required for implementing application history maintenance. The object has three components.

- memory structure of history data constructs
- on disk history management
- method support for history management

4.5.4.1 Configuration of History Data Structure

The JLBS history manager adopts a flexible in-memory data organization for the history records. This ensures support for:

- history maintenance of more than one applications running at a time,
- applications with any type of task interconnections e.g. mesh, tree etc., and
- an efficient data access to the in memory history data.

An example of data organization for in memory history is shown in Figure 4.13. At any moment the history data organization shows the expected communication behavior of the running application as loaded from the previous application runs or the user supplied application task graph[5]. It also contains statistics about the task execution time and communication sizes.

[5] Task graphs shows a rough communication pattern among the application task.

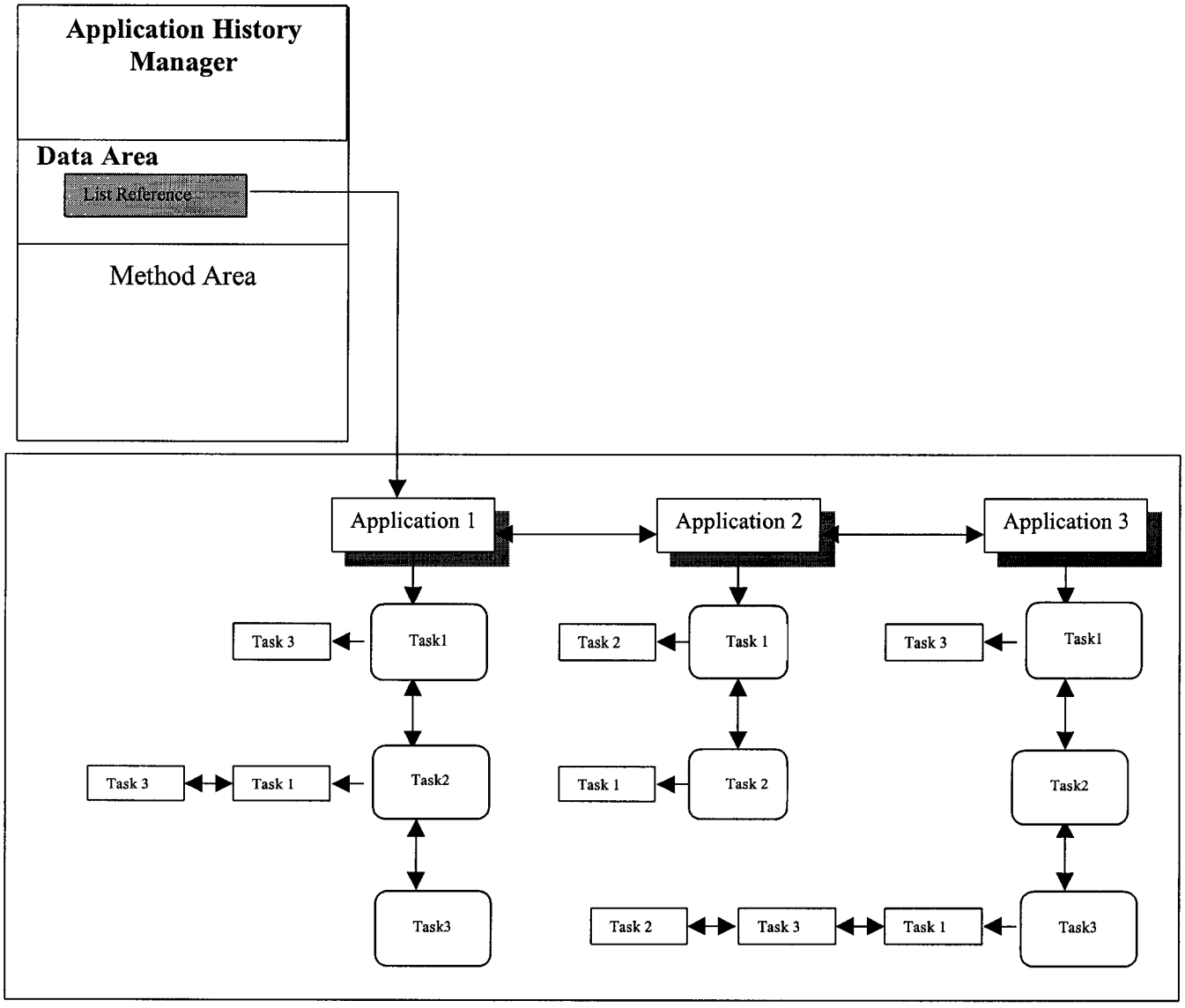


Figure 4.13. Data organization in History Manager Object

4.5.4.2 On disk history management

The object provides mechanisms for on disk history management to support permanent history logs. These logs are updated every time an application runs on the system and will be used during later application runs for estimating the application task behavior. However an easy and efficient repository searching mechanism is one of the key requirements for such a system. This requires that the system should be able to identify unambiguously the application information available in the current history database.

The history object adopts a unique name generation mechanism (Algorithm 4.5.4.2) for every application. These names will be used for saving/ retrieving application history information to and from the on-disk repository. The search can be made even more optimized by maintaining a history search table, containing entries for the applications whose data are available inside the repository. Any reference to the history repository then begins with a reference to this table, which tells about the availability or non availability of application history data. Figure 4.14 shows an on disk image of history table showing the applications available in the history repository.

```
#Record Format
#Application unique characteristics,  user perceived name of the application

#<No. Of Tasks>  <Sum of Sizes of all tasks>  <Name of application>
      5                  32132                      Dummy_Application
      10                 832132                     Matrix_Application
      3                  12132                      Dummy_Application
```

Figure 4.14. On Disk Image For History Search Table

Algorithm 4.5.4.2: Unique id Generation For An Application

Input: User given application name

Output: A unique application id used in application history repository.

Processing:

For (each task included in the application) Do

Count the task in a count variable

Sum up the size of class byte code executable of that class

EndFor

Create the application name using the following formula:

HistoryName = Concatenate task count value and the sum of executable sizes.

Return the computed name.

EndProcessing

Algorithm 4.5.4.2 Unique id Generation For An Application

4.5.4.3 Method Support for history data management

The basic method support can be listed as follows.

- Loading an application history into memory
- Updating an application in-memory history image
- Saving the updated history on disk

The details of algorithms used in implementing the above methods are given below.

Load an application history

The routine is called at the start of an application run by the controller object. In case the application is already running, the history-loading request is denied and results in an error message. The reason for this denial is that in case more than one copy of an application is active, system can not distinctly log the monitoring data for different task copies of the same application.

After confirming the single running instance of the application, the algorithm searches for the application history data (on the permanent storage). If the application has run previously, the data could be located. Otherwise, the algorithm remotely requests for predictable application characteristics from its origin host. One of such characteristics could be an estimate about the communication pattern among the application tasks. The details of the algorithm can be seen in Algorithm 4.5.4.3.1.

Update application history

The routine is needed for updating the application on-disk history with the values in the recent runs. For this, the routine needs to have the task spawn data for each application task. This will help in contacting the hosts, which were used to run one or more task copies. The task agents at those hosts are then contacted and the recent task monitoring data is then gathered from them. The details are listed in Algorithm 4.5.4.3.2.

Save application history

The routine does the job of creating permanent history logs for the recently run application on controller hosts in each domain. It saves the history data for the application in a uniquely identifiable disk file. The unique identifier generated for identifying the application is used as the name for its history data file.

For the fresh runs of an application, the algorithm saves the unique application name in the history repository table. This ensures that the application history file could easily be located next time when it is running. Algorithm 4.5.4.3.23 lists the detailed steps.

Algorithm 4.5.4.3.1: Load an application history

Input: User given application name, name of the application source host

Output: Application history records are loaded in memory from the secondary storage.

Processing:

```
if (Application is not already running)
        Allocate a new history structure in the current history manager object.
        Compute a unique application name used for searching the history repository by using
        application unique characteristics[1].
        Search for the application's existence in the history repository table on the domain control
        host.
        If (application found in history repository) Then
                Load application history data from the history repository
        Else
                Download application communication (task graph) information from the application
                source host
                Build the application communication pattern in history data structures in memory.
                Use default values for other application run statistics e.g. execution time,
                communication data sizes etc.
        EndIf
Else
        Give error message: Cannot run an already running application
EndIf
```

End Processing

Algorithm 4.5.4.3.1 Load an application history

Algorithm 4.5.4.3.2: Update application history

Input: User given application name

Output: In memory history records are updated with the values from the recent run.

Processing:

Extract in memory application history information

For (each application task) Do

Note the task spawn information (generated at the time of task spawn) from the in-memory history records. The information will then be used to gather task monitoring data from remote hosts.

For (each host on which the task runs) Do

Request the task recent run statistics

EndFor

Populate the task history record with the new values

EndFor

End Processing

Algorithm 4.5.4.3.2 Update application history

Algorithm 4.5.4.3.3: Save application history

Input: A running application name

Output: Application history saved on the history repositories in all domains.

Processing:

Check for the application existence inside the in-memory history.

If (application entry not found) Then

    Give error: application is not running

Else

    Create a unique file name for the application history repository.

    Traverse through in memory application task history records and write the data on the history disk file.

If (this application runs first time) Then

    Add the application unique name to the history repository log.

For (every existing domain) Do

    Contact all domain control hosts one by one.

    Send the recently run application history for updating the remote repository.

End For

EndProcessing

Algorithm 4.5.4.3.3 Save application history

4.5.5 Algorithms related with load collection

4.5.5.1 Collecting local host load

Each load agent class collects the current host load on the local system. In the current implementation, load is the average number of jobs in the run queue over the last one, five and fifteen minutes. The detailed view of the algorithm is depicted in Algorithm 4.4.5.1.

***Algorithm 4.5.5.1*: Collecting local load on each domain host**
Input: None

Output: Host status record

Processing:

Collect the local host load average array[1]

Create a host status record. The format for this record is as follows:

- host name
- host load average array
- host speed
- list of running task on this host

Return the host record

EndProcessing

Algorithm 4.5.5.1 Collecting local load on each domain host

4.5.5.2 Give load to domain main controller

This algorithm is used to submit the current system load status to the running main controller. The algorithm first gets the most recent system status, by collecting load from all domain hosts. The recent load array is then sent to the domain main controller which make use of it during balancing decisions.

Algorithm 4.5.5.2: Give load to domain main controller

Input: None

Output: Domain load array with updated load values

Processing:

```
For (all hosts in the current domain) Do
        If (the load agent object is alive at the current host) Then
                Request for the host load status record[1]
                Add the record in the load array of current domain
        EndIf
EndDo
Return the domain load array to domain main controller
EndProcessing
```

Algorithm 4.5.5.2 Give load to domain main controller

4.6 Overhead Assessments

This section discusses the overheads incurred by the distributed programming model of JLBS. As JLBS is built upon the distributed model provided by Java RMI model, it involves all the overheads inherent in the model. The additional overheads it imposes are due to the bookkeeping processing needed to implement load-balancing feature.

4.6.1 Socket Interface

This is the lowest level of support provided for application programmers intending to code distributed applications. This interface has become a standard in contemporary computing. Sockets are analogous to mailboxes and telephones in that they allow users to interface to the network, just as mailboxes allow people to interface to the postal system and the telephones allow access to the telephone system. Their position in the operating system is shown in Figure 4.15

Sockets can be created and destroyed dynamically. Creating a socket means allocating extra memory space, which could hold extra information about current socket connection like the machine and port addresses in use, the type of socket connection which actually implies the protocol used for communication and so on.

The amount of overhead implies here then mainly depends upon the protocol being used for communication. The most common protocols are referred to as the TCP/IP suite of protocols, which provide the following socket types:

- Datagram: These types of socket models potentially unreliable, connectionless packet communication.

- Stream Socket: These model a reliable, connection oriented byte stream.

- Sequenced packet Socket: These model sequenced, reliable, unduplicated connection based communication that preserves message boundaries.

Obviously, the connection oriented socket connections impose more overhead as compared with the others. The overheads caused by socket connections are adequately addressed in [55].

JLBS mainly accesses the transport layer facilities through RMI abstraction, thus it mainly relies on the same socket model on which RMI is based. As RMI make use of HTTP[6] protocol for realizing its services, the socket type used by it is stream based.

In JLBS, there are few situations in which direct socket connections are needed. For example, for the generic daemon object on each host, JLBS uses TCP-based raw socket connections. Due to the required nature of these links, stream based socket connections are used.

4.6.2 JVM

4.6.2.1 JVM Translation Overhead

The portability feature of Java resides in defining a generic machine environment, named JVM (Java Virtual Machine) for different architectures available. Java programs are translated into Java byte code, which could run directly on any JVM without a recompilation. A runtime translation from byte code to native code is done by a Java interpreter. This translation naturally incurs an additional overhead in accessing the available resources. A brief look to this performance degradation was given in Table 4.1

[6] A connection oriented protocol used for web transactions.

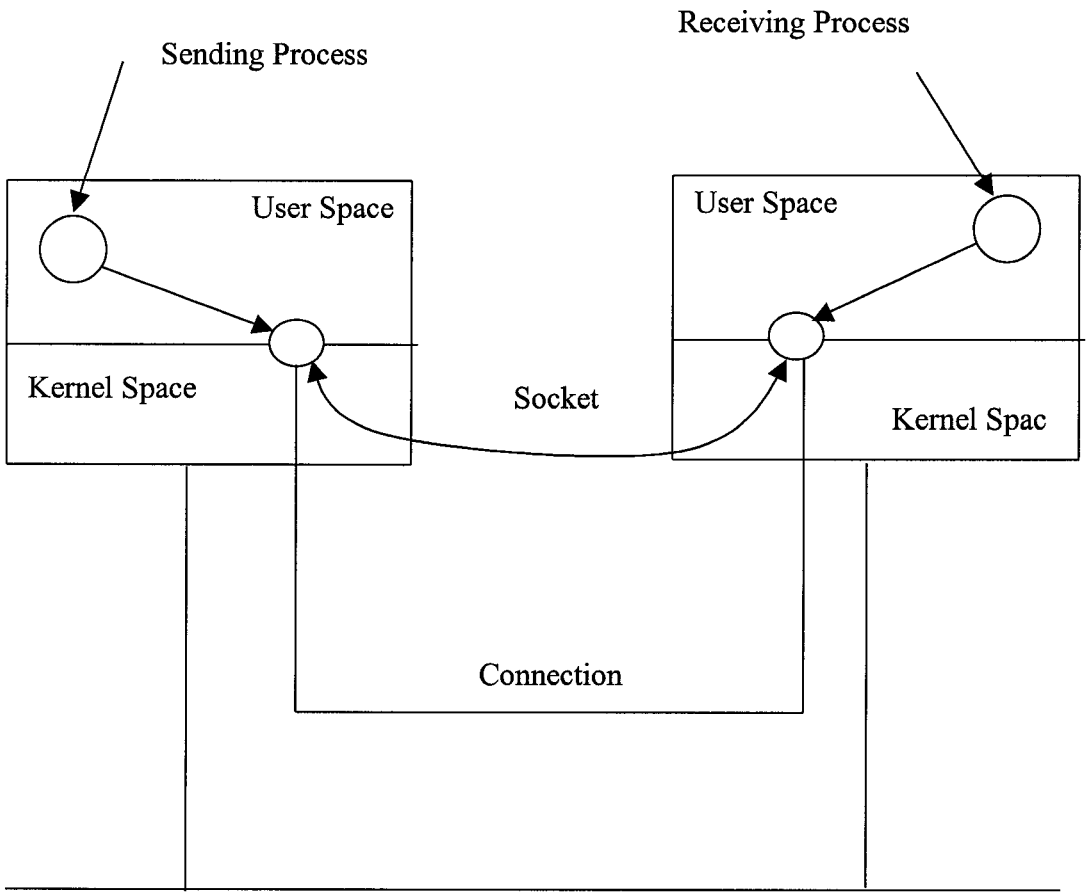


Figure 4.15. Use of Sockets for Inter-process communications

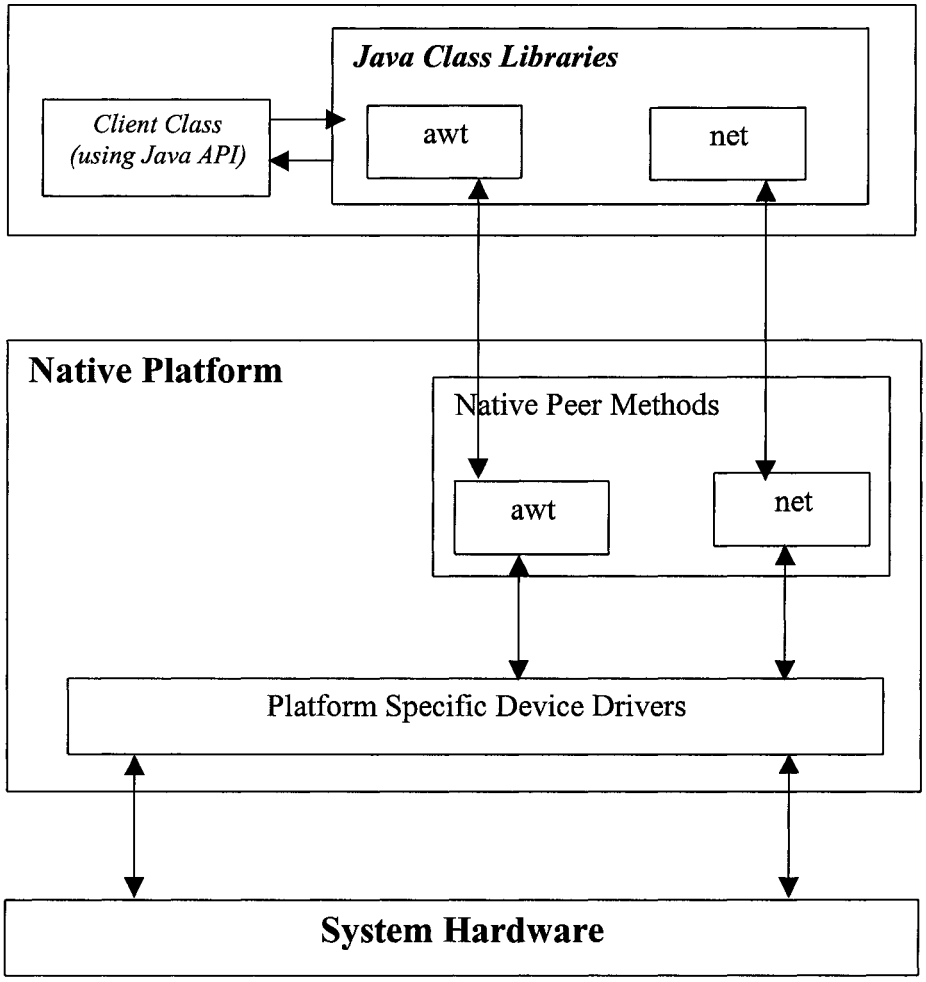


Figure 4.16. Use of Native Methods for Resource Usage Portability

Figure 4.16 shows a scenario of accessing native drivers using Java API. For accessing a native socket structure on some platform, a Java program needs to call a machine specific peer socket class code, assumed to be provided with the architecture's JVM implementation. That peer Java class will then forwards the requested service to the native socket API calls.

4.6.2.2 JVM Thread Scheduling Overhead

Moreover for multithreaded applications, JVM may impose an additional scheduling overhead. However the performance and behavior of Java thread layer is directly dependent upon the nature of the underlying platform used (threaded or non-threaded) and the way Java threads are mapped over the native system threads. For platforms, like Mac OS and some older versions of UNIX, which do not support thread requires JVM to emulate threads on them. With platforms having native threading capabilities, such as Solaris and Windows NT, JVM maps Java threads to native threads. The implementations of this mapping are again transparent to the users and could be one of the following.

- One-to-one mapping (as on Windows NT system)
- Many-to-one/Many-to-Many mapping (as on the Solaris system)

Yan et.al. [8] discusses the performance of Java threads on two different multithreaded platforms: Windows NT and Solaris. They used a well-known compute-intensive bench mark, the EP benchmark, to examine various performance issues involved. Results showed that the performance of Java threads differs depending on the various mechanisms used to map Java threads on native system threads and the scheduling policies for these native threads.

4.6.3 RMI

RMI abstraction imposes additional overheads due to additional mechanisms:

4.6.3.1 Stub/skeleton translation layer

This layer does the job of data marshalling/ unmarshalling for the method parameters or return values. A stub code provides proxy method calls for each exported remote method in its corresponding remote object. Thus a remote method invocation by the client, results in an additional transparent invocation of an equivalent proxy method in the stub code.

Moreover, the stub/skeleton layer accounts for providing a logical messaging structure between communicating objects. This entails the use of additional buffers and routines to manipulate these buffers for fragmenting, encoding, sending and receiving the messages.

4.6.3.2 Remote reference layer

The remote reference layer is responsible for understanding what a particular remote reference means. A remote reference may refer to a remote object which may reside on a remote machine, on the same local machine (in a separate JVM) or may be on multiple machines. In essence, the remote reference layer translates the local reference to the stub into a remote reference to the object on the server, whatever the syntax and semantics of the remote reference may be. This control data is then passed to the socket layer for implementing the real communication.

4.6.3.3 Dynamic downloading of stub codes

The basic requirement for a client to access a remote object, is to have the object's stub class code. An instance of the stub object will then be used to receive the client calls for remote invocations, which will then properly translated to the target remote object. If the code is available locally at client, it only involves an overhead of class instantiation.

Otherwise a dynamic downloading of class code has to be performed before making use of it[7]. This extra communication will account for additional delays in the method invocation, but lesser effect during the execution.

4.6.3.4 Concurrent method invocations

For a remote object, there is always a possibility of having a method called by more than one client at a time. The current RMI specifications give no clear definition of the type of such invocation [62]. They may be multithreaded or serialized to each other. In any case, such situations impose a traffic (or congestion) overhead for a client request.

4.6.4 JLBS

4.6.4.1 Tests for Application Integrity

In order to have robust system integrity, all objects running in the RMI environments should have some means of ensuring each other aliveness. RMI checks this integrity only at the time of a method invocation, by using a PING request [62]. However, a complete integrated system view is needed at all times. JLBS does this by using periodic dummy method calls on each object. The overhead imposed by such calls could be considerable in case of large system sizes. Localization through distributed domains reduces this effect.

4.6.4.2 Application History Maintenance

This is a major source of overhead in JLBS as all associated overhead is in some way or other related to this issue. The driving factor behind this is to get the history information transparent to the user.

- Generation of a unique identification code for every parallel application.
- Search for an application on the on-disk history repository.

[7] Currently, RMI supports this dynamic downloading by make use of an http request to the server host.

- Read history data from the secondary storage.
- Retrieve recent application run behavior from every host, having executed this task.
- Log the updated history records on the secondary storage.

4.6.4.3 Application Task Monitoring

On every host, the running task characteristics are kept under monitoring by some JLBS module. The main characteristic needed is the volume of communication a task experiences with other tasks. In other words, if a remote method of task X is invoked by another task Y, the sum of the size of method parameters and its return value will give the total communication volume takes place.

The JLBS task monitoring interrupts every remote invocation in order to get the relevant monitoring information including the name of the tasks involved and the size of communication volume involved. After extracting this information, the method invocation will be resumed. This process results in an additional overhead with every remote invocation.

4.6.4.4 System Load Monitoring

This issue is another candidate for the most responsible activity for imposing overhead. The overhead mainly depends upon the frequency of load collection performed by the system. For n hosts every collection involves the generation of n messages. However, in JLBS the effect of the overhead is being limited by adapting a domain based hierarchical load collection system.

CHAPTER 5

Experimentation and Results

This chapter discusses the performance analysis of the current JLBS implementation. The objectives of the experimentations are

1. to show the scalability of JLBS system under different load environments,
2. to compare the balancing strategy used in JLBS with other alternative strategies discussed in chapter 3 (round robin and random assignments),
3. to report the degree of overheads imposed by JLBS software layer.

The characteristics used in the experimental analysis are application completion times, system overhead , speedup and system scalability. The exact definition of each of them are as follows.

- The Completion time of an application is the wall clock time, which is measured from the submission of its first task till the completion of the last task.

- System overhead represents the execution cost results due to system bookkeeping activities. This metric then could be used to classify applications to run on the JLBS platform.

- System speedup is measured by making use of application completion times. Generally speaking, speedup is the ratio of the sequential completion time of an application (using one machine) to the parallel execution (using n-machines, with or without load balancing time).

$$\text{SpeedUp} = \frac{(\text{Application Completion Time})_{\text{Sequential}}}{(\text{Application Completion Time})_{\text{Parallel}}}$$

However, two possible approaches could be used in speedup computations depending upon the computation formulae used for completion times.

- System scalability is defined as the improvements attained in the system speedup with the inclusion of improved system configuration.

The application test suite, used in the analysis, is composed of both generic and real applications. Generic applications allow representing main types of parallel applications such as CPU, IO or communication intensive. They also allow the reconfiguration of an important task characteristic, the task granularities.

5.1 Generic Applications

Rigorous testing of a framework for parallel and distributed computing cannot be achieved using real applications. Each application is different and possesses own limits and cannot easily be reconfigured. That's why generic application based testing is more practical. Tasks in a generic application can be reconfigured or tuned easily as one of the following classes.

- CPU intensive
- IO intensive
- Communication intensive

5.1.1 CPU Intensive

Purely CPU intensive tasks are independent programs, which execute as a single process. Each process will be a standalone entity, in that it has no communication/interaction with any other processes currently running on the system and spends most of its time in doing some computation. The amount of time, such processes take, totally depends upon the amount of computation power available.

For the JLBS experiments, the CPU intensive tasks use a long computation loop, doing some complex number crunching operations. The number of loop iterations determines the task granularity and is controllable by making use of application specification files. Figure 5.1 shows a representative example of CPU intensive task loop.

```
Algorithm: CPU-intensive task loop
Input: None
Output: Runs CPU-intensive loop
Processing:
Begin
        Read task granularity index (from application specification file).
        Calculate task iterations (for CPU-intensive), using granularity index.

        For (number of iterations)
                CPU-intensive number crunching operations
        End For
End
```

Figure 5.1 A CPU intensive task loop

5.1.2 Independent IO Intensive

IO intensive tasks are processes, which do not involve communication among them and have only IO. A heavy percentage of its time is spent in reading data values from one or more files and generating the results in the same/different files.

Again the IO granularity will determine the amount of such processing performed which can be configured using application specification file. The loop being used for such tasks in JLBS experiments is shown by Figure 5.2.

```
Algorithm: IO-intensive task loop
Input: None
Output: Runs IO-intensive loop
Processing:
Begin
        Read task granularity index (from application specification file).
        Calculate task iterations (for IO-intensive), using granularity index.
        Open files for reading/writing data.

        For (number of iterations)
                Read/write string data from/to files
        End For
End
```

Figure 5.2 An IO Intensive Task Loop

5.1.3 Communication Intensive

This class comprises of parallel tasks communicating with each other to produce the desired results. Generally, a parallel application is composed of two or more instances of such tasks communicating among each other with some communication patterns.

For JLBS experimentation, tasks having generic communication patterns are used. These tasks imitate heavy communication by exchanging large amount of data (numeric/string) among each other. Again the task granularity characteristic defines the degree of this communication intensity. Code fragment, shown in Figure 5.3, is used for simulating a communication intensive task.

```
Algorithm: COMM-intensive task loop
Input: None
Output: Runs COMM-intensive loop
Processing:
Begin
        Read task granularity index (from application specification file).
        Calculate task iterations (for COMM-intensive), using granularity index.

        For (number of iterations)
                Read data from input task links.
                Write data to output task links.
        End For
End
```

Figure 5.3 Communication Intensive Task Loop

5.2 Real Application

Generic applications are helpful in creating hypothetical environments, which could be controlled in terms of different application characteristics. However executing a real application allows observing the system performance under more real application conditions.

A real application generally involves a mix of IO, computation, and communication. For example, in case of matrix multiplication IO activity constitutes of reading the input matrices from input files and writing the results to output files, communication is needed during data partitioning and gathering of results from multiple workers (tasks) and computation is done locally by each workstation. However the nature of this application is close to CPU intensive.

The definitions for both sequential and parallel matrix multiplications are given in the following subsection.

5.2.1 Sequential Matrix Multiplication

In sequential execution, matrix multiplication is done as a single process. The resultant matrix is computed using 3 cascaded for-loops. Each iteration results in computation of one element of resultant matrix.

$$C = \sum_{i=1,m} \sum_{j=1,m} \sum_{k=1,m} A_{ik} * B_{kj}$$

For example, for two matrices A and B each of size (m x m), product matrix C could be computed as:

5.2.2 Parallel Matrix Multiplication

In parallel case, the problem constitutes of one server and N clients (worker) processes. The server has the responsibility of initiating the clients and partitioned matrix input data among them. The clients perform multiplication on their own datasets and communicate the results to the server. The server gathers the results and writes them to a file.

Again for two MxM matrices (A & B) and N clients, each client computes (M/N) rows of the resultant matrix (C). Following is the computation done by nth worker, W_n.

$$C_{Wn} = \sum_{i=1,\ M/N} \sum_{j=1,\ M} \sum_{k=1,\ M} Aik * Bkj$$

The complete result will then be computed by integrating the results from all workers as follows.

$$C = \sum_{n=1,N} C_{Wn}$$

5.3 Experimental Setup

- All experiments were performed on a cluster of 14 machines LAN (196.1.67.0). The network is a 10Mbps Ethernet. The system allows the use of the cluster as one or multiple balancing domains, depending upon the environmental requirement. The

experimentation phase, however, make use of these machines as a single balancing domain.

- The details of the machine configurations used in the environment are as follows:
  - All 14 machines are Intel based Pentiums with speeds 233 MHz each and 64 Mbytes of memory.
  - 13 machines are running Windows 2000 while one is having a Linux (kernel version 2.0.31) on it.
  - The Java Development Kit used is JDK1.2 with the use of just-in-time compiler.

- Four different applications are used.
  - Application 1 - 10 tasks each having a different number of instances which adds up to 24 instances, all CPU Intensive
  - Application 2 - 10 tasks each having a different number of instances which adds up to 24 instances, all IO Intensive
  - Application 3 - 16 tasks each having a single instance, all communication Intensive
  - Application 4 - Real Application (Matrix Multiplication)

Each test comprised of at least 3 application runs. In each run, the completion times for both individual tasks and the complete applications are determined.

- The tasks granularities are expressed in terms of integers used in application specification files. The higher the number more the granularity.

- All the experiments were conducted under two typical run-time environments having
  - A fixed background load, with all workstations running same external load
  - A variable background load, with every host is imposed upon a different external load.

The load is created as separate JVM windows each running an infinite loop.

5.3.1 Task Granularity Details

- In terms of the task granularities, all tests are performed under two typical cases:
  - Each application task with the same granularity, for instance 10
  - A random granularity distribution is used for the application tasks. The standard deviation for the distribution was taken high enough to represent the real situations. Table 5.1 shows the exact figures used for this distribution.

Tasks	Granularity
Task 1	55
Task 2	6
Task 3	36
Task 4	54
Task 5	23
Task 6	55
Task 7	43
Task 8	30
Task 9	9
Task 10	23
Standard Deviation	18.35

Table 5.1 Granularity Distribution used for variable granularity applications

5.3.2 Experiment Configurations

The names and configurations for the experiments performed are as follows:

1. System overhead testing
2. System scalability testing
3. System performance comparisons with different balancing strategies

5.3.2.1 System Overhead Testing

While performing a load balanced application execution, JLBS exhibits a tangible amount of execution overhead. It results due to many bookkeeping activities necessary to maintain a load-balanced environment. These activities may include

- Periodic collection of system load
- Periodic integrity testing for the system components &
- Periodic logging of book-keeping data on the local storage.

Formula used for Overhead Computation

JLBS overhead could be estimated by comparing the sequential and a single machine parallel runs (on a single machine) of an application. This requires the definitions of the following two application times.

- Sequential execution time (*Se*)
- Local execution time (*Le*)

Sequential execution time (*Se*)

Se corresponds to the application time while running as a single thread of control, without a JLBS environment. Thus it purely reflects the time taken by the sequential run[1]

Local execution time (*Le*)

Le corresponds to the application time when its tasks (processes) get executed all on the same machine on which they are initiated, without any balancing. Execution is done on a single machine. However as all the JLBS components are active and the application agent task is communicating with them, the running time involves the overhead due to the JLBS system bookkeeping activities.

In addition to the system overhead, the application completion time involves the context switching overheads, due to time-shared nature of the underlying platform. However, this

---

[1] To eliminate the possibility of context switching overhead, no external process runs are assumed.

overhead could be avoided if applications with just a single task component are used i.e. an application with one application agent and a single application task.

JLBS overhead can then be properly calculated as the difference of the above two time values.

$$JLBS\ Overhead = (Le)_{1T} - Se \qquad (5.1)$$

where

$(Le)_{1T}$ = Le for a single task application.

5.3.3 System Scalability Testing

System scalability tests exhibit the improvements in system performance as a direct consequence of the changes in the system and application size.

- The system performance measure used in these tests is the speedup.
- In order to avoid the inclusion of any other factor, only CPU intensive application is considered in the testing.
- The scalability is shown by using the following three approaches:
  - *First Approach*: It involves running applications with different task sizes on a fixed host configuration. Currently, applications with 4, 8, 12 and 24 tasks are being considered with 4 hosts in the system. Scalability is proved if the speedup increases with application size or at least not decreased after a threshold speedup value.
  - *Second Approach*: In this approach, two equal size applications (having same number of tasks) run concurrently on certain host configuration. Their parallel completion times are then compared with an application (run on the same host

configuration) having a size equal to the sum of two application sizes. Scalability will be proved if the smaller applications, running concurrently with each other, show better (or at least similar) completion times as compared with the application of larger size.

- *Third Approach*: This involves the running of an application of a certain size on different host configurations. Currently used configurations are 4, 6 and 8 hosts with a 24 task application size. Here scalability will be proved if the speedup is shown to have a consistent increase with the increasing number of hosts in the system.

5.3.4 System Performance Comparisons with Different Balancing Strategies

This experiment is designed to compare the performance of the proposed balancing heuristic with two other strategies named: *Round Robin* and *Random* assignment strategies. The details of these strategies can be seen in chapter 4 (section 4.5.2.3)

- System speedup is again taken as the performance measure.
- Both generic (all three classes: CPU, IO and communication intensive) as well as a real application (matrix multiplication) are used.

5.3.5 Formulae for Speedup Computations

Theoretically, speedup represents the effect in execution speed of an application when it is executed on *N* machines as compared with a single machine run. Thus it is the measure of improvement in an application execution time when run as collection of parallel tasks.

$$SpeedUp = \frac{(CompletionTime)_{Single\ -\ machine}}{(CompletionTime)_{N\ -\ machines}}$$

However, while computing the speedup, there could be two different approaches for finding application completion times. Each of them introduces a different speedup computation approach as defined below.

Speedup - S1

A parallel application consists of many parallel tasks (processes). Each application task has its own completion time, defined as the clock time between its start and end. Thus the application completion time could be computed as the average of completion times for all its parallel tasks. This could be applied both when the application runs on a single machine and when running on n-machines.

Suppose for an application having n parallel tasks (processes),

- $(TsSm)_i$ = Task completion time for task i, running on a single machine
- $(TsNm)_i$ = Task completion time for task i, running on N machines, with load balancing.

Then speedup S_1 can be defined as,

$$S_1 = \frac{\frac{\sum_{i=1,n}(TsSm)_i}{n}}{\frac{\sum_{i=1,n}(TsNm)_i}{n}} \qquad (5.2)$$

This approach provides a practical speedup for a timesharing system like Unix and Windows NT. However, theoretically it does not provide a correct speedup, as it includes the effect of timesharing depending on the order the processes enter the system.

Speedup - S2

To estimate the theoretically correct speedup, sequential execution time for a parallel application has to be computed. It could be defined as the time taken by the best known serial equivalent of the given parallel application. This step may not be practical for most of the applications as the best serial program execution time is not known, or else the parallel algorithm differs so greatly from the serial algorithm that a comparison of their times is meaningless. However for applications in our test suite (generic and matrix multiplication), equivalent serial versions could easily be found by simply concatenating task loops.

The parallel application execution time is computed as the difference between the first process entry time to the last process termination time.

$$S_2 = \frac{S_e}{(Last\,\mathrm{Pr}ocessEndingTime - First\,\mathrm{Pr}ocessEntryTime)} \qquad (5.3)$$

5.4 Results and Discussion

This section discusses the results obtained by performing experiments on the previously defined experimental setup.

5.4.1 System Overhead Testing

The overhead of JLBS is observed by comparing the sequential and local execution times, as defined by Eq(5.1). Table 5.2 and Figure 5.7 shows the completion times of an application having a single task (CPU Intensive, granularity index = 50), in three different runs. The following could be observed.

- For the parallel run on a single machine (Le), the runs are done within a same JVM. A separate class (named jlbs.tests.JlbsExperiments) used as a loader for running the test application. The loader after loading the application's class bytes, invokes its run method. At the completion, the application completion time is then logged in the loader class.
- The runs are repeated 3 times but all the subsequent runs (after the first one) don't need reloading of application class bytes in memory. Due to this reason the first run shows a comparatively higher value (563 seconds as compared with the other twos (both 562 seconds).

Run No.	*Sequential Run Se (sec)*	*Single Machine Parallel Run*	*Difference*	*% Difference*
1	554	563	9	1.62%
2	554	562	8	1.44%
3	554	562	8	1.44%
Average	504	562.33	8.33	1.5%

Table 5.2 JLBS Overhead Estimation

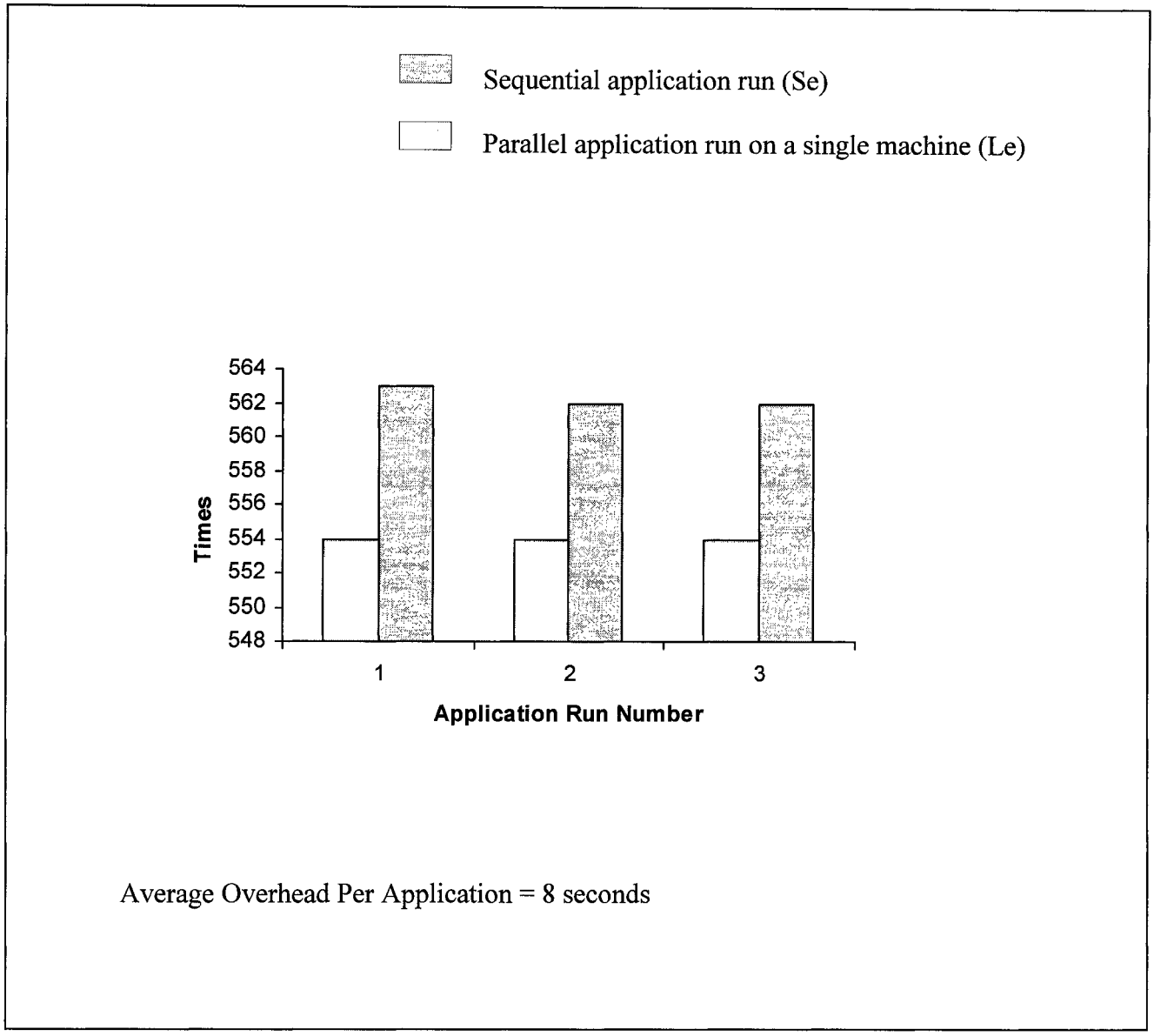


Figure 5.4 JLBS Overhead Estimation

This overhead may result due to the existence of system book-keeping activities like:

- interaction of application controller task with various system modules, and
- interactions among the system modules for certain book-keeping activities like periodic load collection, periodic integrity testing among system agents running on different machines etc.

The result implies the suitability of running high granularity applications i.e. application whose completion times are much greater than 8 seconds for the JLBS system. This is because the longer the application is, the smaller will become the percentage of overhead results as compared with its sequential run.

5.4.2 System Scalability Testing

First Approach:

This approach uses a fixed host configuration of 4 hosts and 4 different application runs with sizes 4, 8, 12 and 24 tasks (all CPU intensive) respectively. The external loads on hosts is assumed to be constant and the speedup is computed using formula for S1 (Eq. 5.3).

Table 5.3 is giving the test results for three applications run. Figure 5.4 is plotting them as a column chart. The results clearly show that the system is highly scalable as no serious degradation in speedup with the increase in the application size is observed. The maximum speedup achieved with the 4 task application is 3.8 which is very close to the ideal value of 4.0, for 4 host configuration. As the application size grows, the speed up

tends to increase or at least fixed at this value. This proves the system to be highly adaptable for increasing levels of system load.

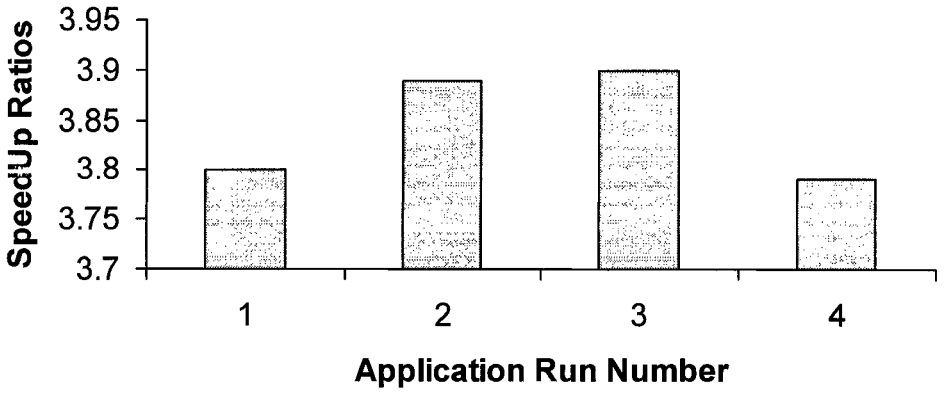
S. No.	Application Configuration (Number of tasks)	Average Sequential Times (sec)	Average Parallel Times (sec)	SpeedUp
1	4	468	123	3.8
2	8	942	242	3.89
3	12	1405	360	3.9
4	24	2813	742	3.79

Table 5.3 System Scalability Testing – Different Application Sizes on Fixed Host Configuration

Second Approach:

The approach uses three different application runs and then compares their parallel completion times. At first an application with 8 tasks (all CPU intensive) is run on the system with 4 hosts. Then 2 applications with 4 tasks each (all CPU intensive) run concurrently on the same host configuration. The external loads on the hosts is again fixed and the speedup is computed using formula for S1 (Eq. 5.3).

Table 5.4 shows the test results achieved in this approach. The results reveal that the concurrent run of two applications does not impose any additional system overhead. In the test the completion time for an 8 task application (241 seconds) is shown to improved when run after decomposing into two equal sized applications of 4 tasks each (197 & 194 respectively). This shows that the further parallelization of any application would add up



Host Configuration Used = 4 hosts

Speedups Achieved

Application Size	Speedup
4 tasks	3.8
8 tasks	3.89
12 tasks	3.9
24 tasks	3.79

Figure 5.5 System Scalability Testing – Different Application Sizes on Fixed Host Configuration

to improved execution performance even in the similar load conditions[2]. This also indicates that concurrently increasing the number of applications has no degradation effects on system performance. This again proves the system scalability.

S. No.	Application Size (# of Threads)	Average Sequential Times (sec)	Average Parallel Times (sec)	SpeedUp
1	8	942	241	**3.91**
2	4	468	197	**2.38**
3	4	468	194	**2.41**

Table 5.4 System Scalability Testing – Variable number of Applications on Fixed Host Configuration

Third Approach:

In this approach, the system scalability test will use a CPU intensive application (24 tasks) runs on three different system host configurations with 4, 6 and 8 hosts respectively. The speedup is measured using formula for S1 (Eq. 5.3). Three application runs are done for each host configuration.

To have a rigorous scalability test under all possible load conditions, the experiment is repeated four times for the following environmental cases.

Case A: Fixed load on each host, application with equal task granularities

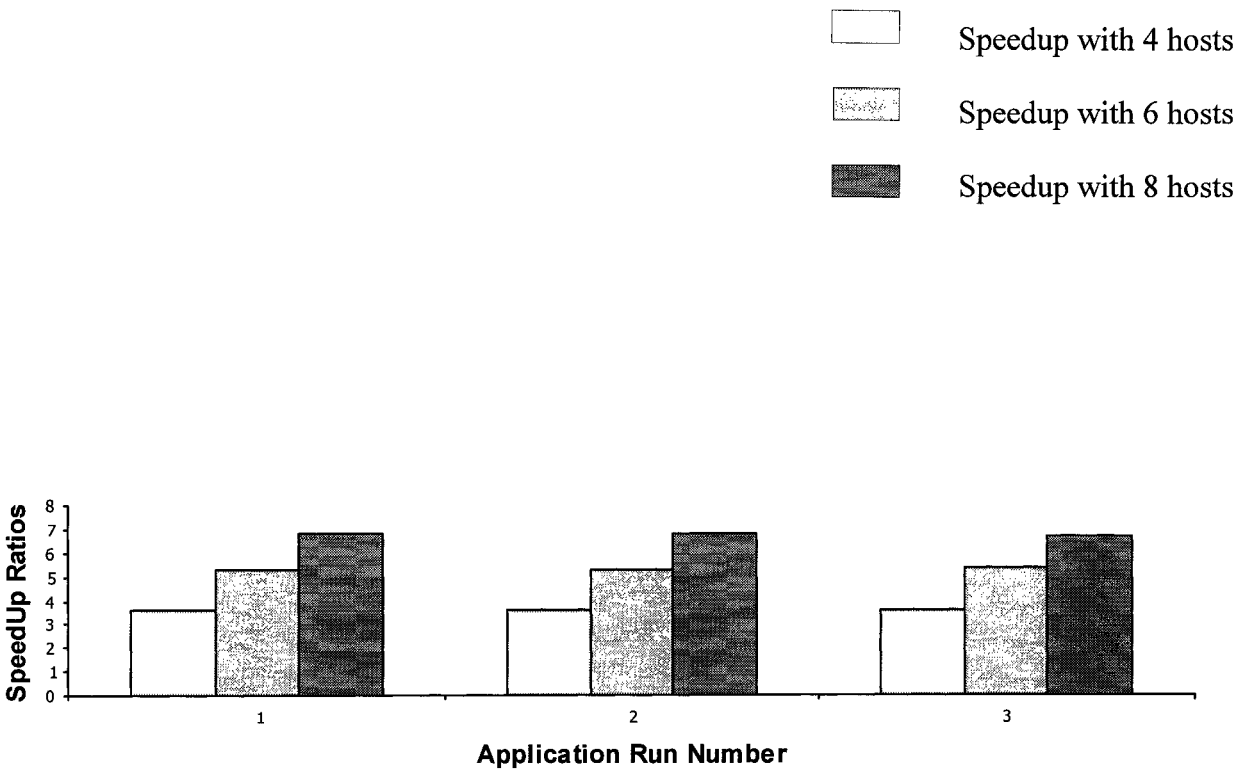
Table 5.5 and Figure 5.6 shows the application completion times for the environment of Case A. The speedup values shows clear improvements in the system performance with the increase in number of hosts. Also due to the consistent nature of the environment used, all values are very close to each other. The average speedups for 4, 6 and 8 hosts are respectively 3.64, 5.32 and 6.79.

Due to the presence of system book-keeping overheads, the speedups attained are understandably lower than the ideal speedups (4.0, 6.0 and 8.0 for 4, 6 and 8 hosts used respectively). However the percentage decrease of overhead as compared with the ideal value, seems to be increasing with the increased number of hosts. In case of 4 hosts it is 91% which is other twos which are 89% and 85% respectively. Again this behavior is something understandable as with the increased system size the overhead of system book-keeping gets increased also.

S. No.	Sequential Run	4 Hosts		6 Hosts		8 Hosts	
	Se (sec)	Time (sec)	Speed Up	Time (sec)	Speed Up	Time (sec)	Speed Up
1	2313	636	3.64	439	5.27	337	6.86
2	2313	636	3.64	438	5.28	341	6.78
3	2313	635	3.64	428	5.4	343	6.74
Average			3.64		5.32		6.79
% of Ideal			91		89		85

Table 5.5 Scalability Testing - Application completion times for Case A

[2] 2 tasks per host in our case.



Average Speedups

4 Hosts: 6 Hosts: 8 Hosts :: 3.62 : 5.32 : 6.79

Figure 5.6 Scalability Testing - Application completion times for Case A

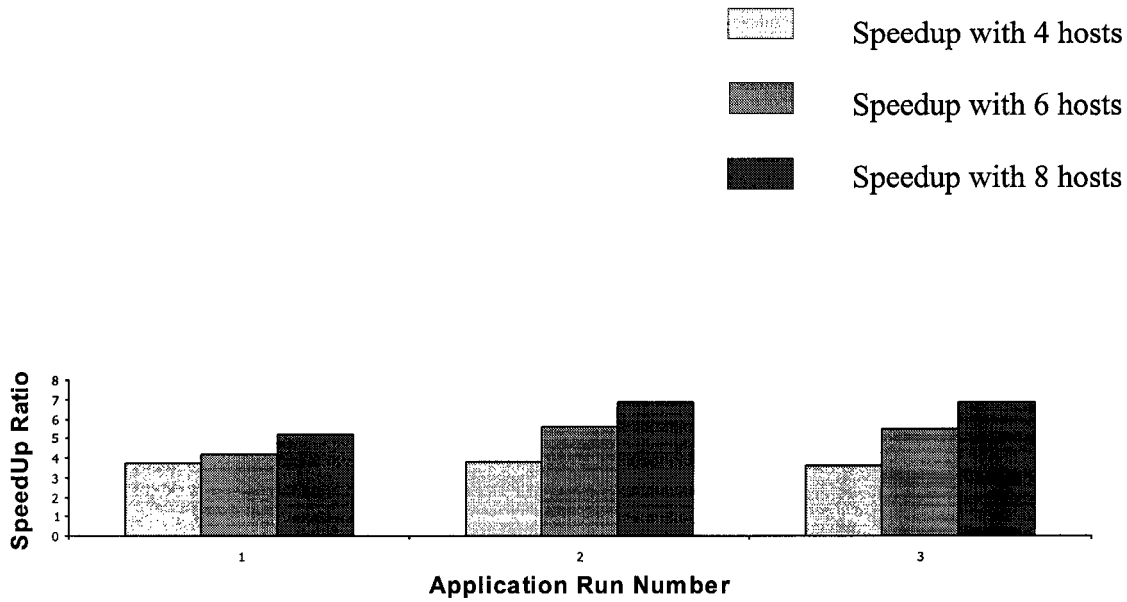
Case B: Fixed load on each host, application with different task granularities

Table 5.6 and Figure 5.7 shows the application completion times for the environment of

S. No.	Sequential Run	4 Hosts		6 Hosts		8 Hosts	
	Se (sec)	Time (sec)	Speed Up	Time (sec)	Speed Up	Time (sec)	Speed Up
1	8209	2204	3.72	1984	4.14	1568	5.24
2	8209	2159	3.8	1482	5.54	1184	6.93
3	8209	2254	3.64	1489	5.51	1195	6.87
Average			3.72		5.06		6.35
% of Ideal			93		84		79

Table 5.6 Scalability Testing - Application completion times for Case B

Case B. Here again the same behavior of improvement in speedup can be seen with the improved configuration used. The average speedups attained for 4, 6 and 8 host configurations are 3.72, 5.06 and 6.35 respectively. However due to the involvement of some variability in the application (tasks with variable granularities with a considerable deviation among them), the first run in all cases shows some inferiority in terms of the completion times. This inferiority results because of the unavailability of application history data at the time of the first run and thus a wrong assumption of constant task behaviors at that time. This also explains why this effect could not be felt in the pervious case (Case A) where all tasks were having the same run behavior in reality.



Average Speedups

4 Hosts: 6 Hosts: 8 Hosts :: 3.72 : 5.06 : 6.35

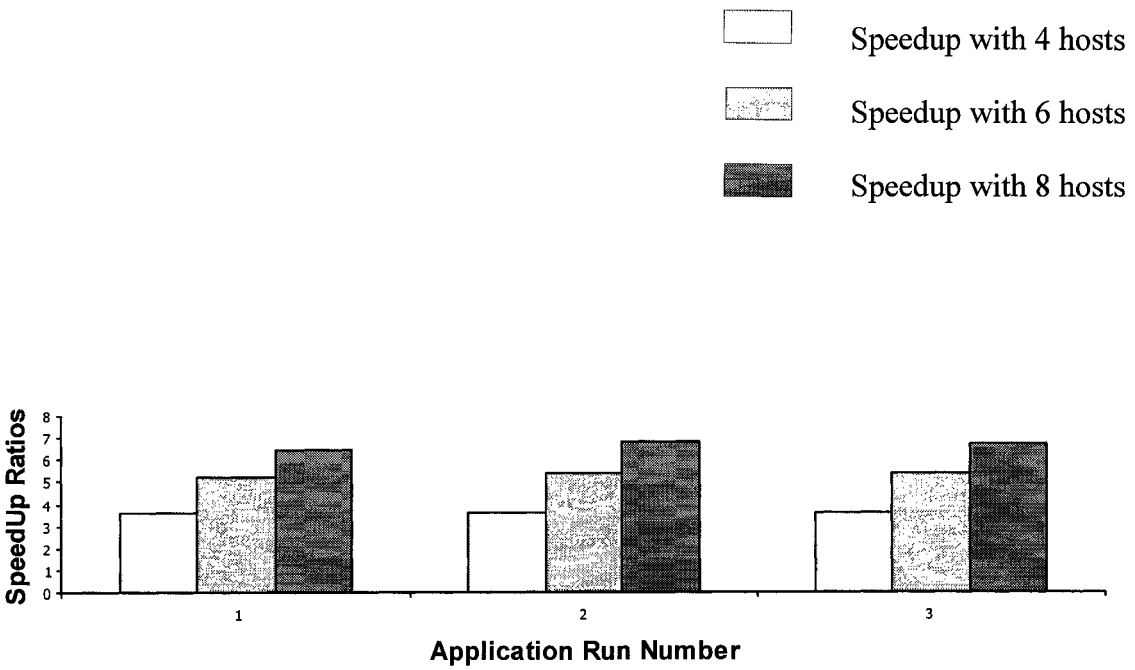
Figure 5.7 Scalability Testing - Application completion times for Case B

Case C: Variable load on each host, application with equal task granularities

Table 5.7 and Figure 5.8 shows the application completion times for the variable host loads with fixed task granularities in the application. The system behavior is still shown to be scalable as the speedup averages are 3.62, 5.32 and 6.7, for 4, 6 and 8 host configurations respectively. Also, the runs show similarity with the ones in Case A in terms of the application's first run behavior i.e. the completion times obtained in first runs are not showing much inferiority. This is again due to a match with the system assumption and the fact that all tasks have the same behavior in terms of their granularities (similar granularity tasks).

S. No.	Sequential Run	4 Hosts		6 Hosts		8 Hosts	
	Se (sec)	Time (sec)	Speed Up	Time (sec)	Speed Up	Time (sec)	Speed Up
1	2313	644	3.59	441	5.24	356	6.5
2	2313	639	3.62	432	5.35	339	6.82
3	2313	635	3.64	431	5.37	341	6.78
Average			3.62		5.32		6.7
% of Ideal			91		89		84

Table 5.7 Scalability Testing - Application completion times for Case C



Average Speedups

4 Hosts: 6 Hosts: 8 Hosts :: 3.62 : 5.32 : 6.7

Figure 5.8 Scalability Testing - Application completion times for Case C

Case D: Variable load on each host, application with variable task granularities

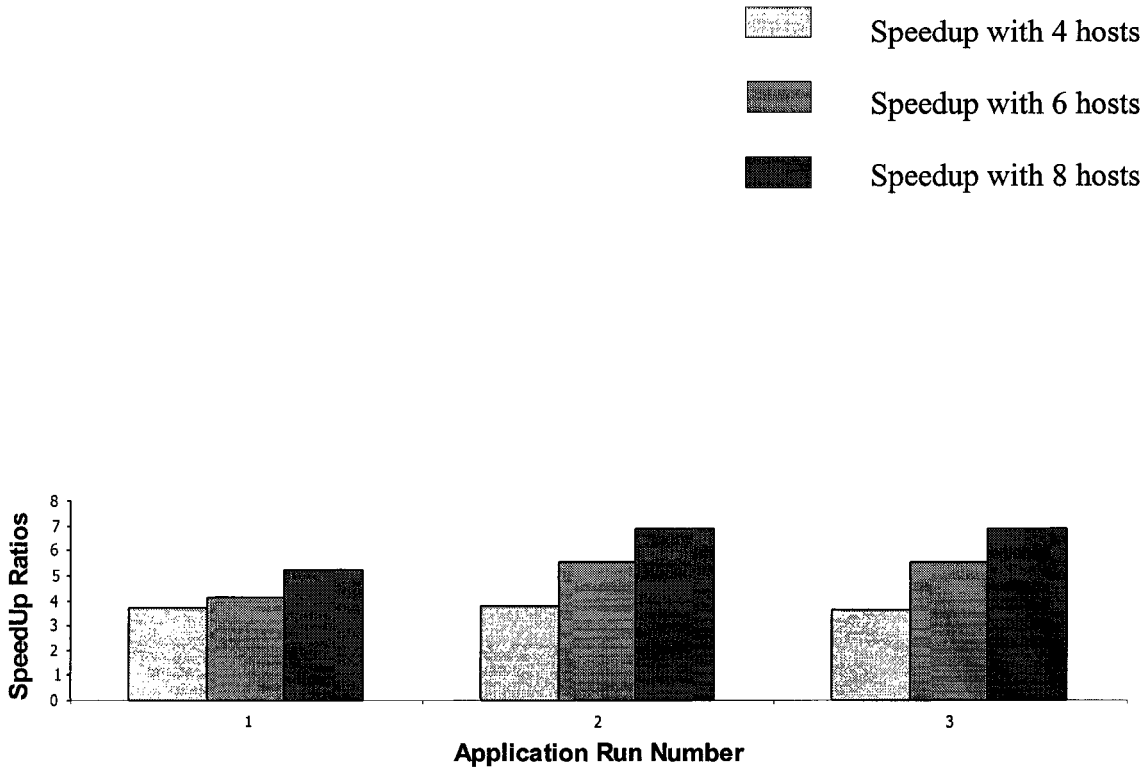
Table 5.8 and Figure 5.9 shows the application completion times for case D. Again, the average speedups (3.66, 4.57 and 6.14 for 4, 6 and 8 hosts respectively) are proving the system scalability feature. Due to the wrong assumption about task granularities at the time of first run, the inferiority of the first run is very obvious in all three configurations.

S. No.	Sequential Run	4 Hosts		6 Hosts		8 Hosts	
	Se (sec)	Time (sec)	Speed Up	Time (sec)	Speed Up	Time (sec)	Speed Up
1	8209	2308	**3.56**	2017	**4.07**	1579	**5.2**
2	8209	2210	**3.71**	1698	**4.83**	1203	**6.82**
3	8209	2219	**3.7**	1702	**4.82**	1281	**6.41**
Average			3.66		4.57		6.14
% of Ideal			92		76		77

Table 5.8 Scalability Testing - Application completion times for Case D

5.4.3 System Performance Comparisons with Different Balancing Strategies

This experiment test the merits of the proposed balancing heuristic (section 4.5.2.1) in comparison with the two alternative strategies named Random and Round Robin



Average Speedups

4 Hosts: 6 Hosts: 8 Hosts :: 3.66 : 4.57 : 6.14

Figure 5.9 Scalability Testing - Application completion times for Case D

assignments(sections 4.5.2.2 & 4.5.2.3 respectively). The host configuration used is 8 hosts. The test uses both generic and real applications with all 4 cases of environment combinations (Case A - Case D) as did in the previous experiment. For the generic application case, all 3 possible types are considered. They include:

- Application with all tasks CPU-Intensive in nature.
- Application with all tasks IO-Intensive in nature
- Application with all tasks communication intensive in nature, communicating in a tree like pattern among each other.

For the real application case, matrix multiplication application is being considered. The tests are carried out with different matrix sizes, in order to get the size having suitable computation to communication ratio. The size is then chosen to test with other strategies.

5.4.4 Generic Applications

5.4.4.1 CPU Intensive Application

Table 5.9 to 5.12 shows the application completion times for CPU intensive application obtained with different assignment strategies, for case A through case D respectively. Application contains 10 tasks (all CPU intensive) with different number of instances, which adds up to 24. Three runs of the application will be used for each of the four cases.

Case A: Fixed load on each host, application with equal task granularities

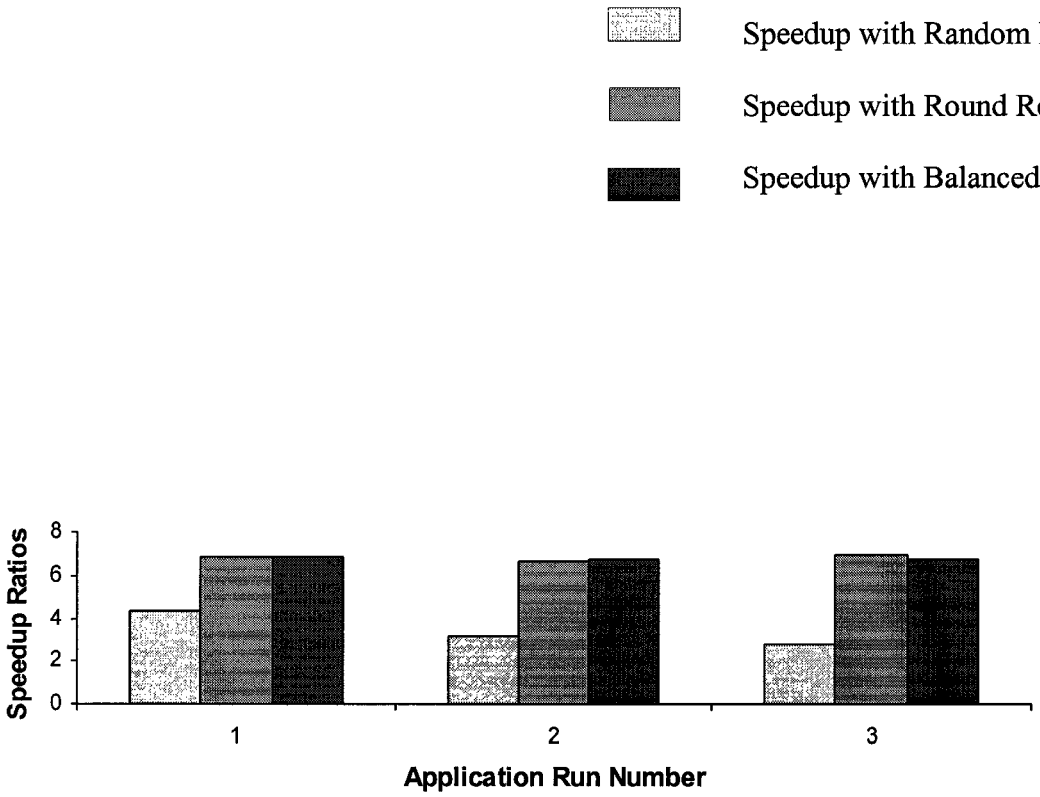
The completion times in Table 5.9, sketched in Figure 5.10, clearly shows the inferiority of random assignments (average speedup = 3.41) as compared with the other twos. Also due to the randomness involved, completion times obtained in random assignments are considerably different in all three runs.

However, balanced and round robin assignments show very similar behavior in all three runs. This could easily be explained, as the environment and application conditions used (Case A) implies for this similarity. Due to the fixed load on all hosts and equal task granularity the fixed assignment could provide the best-balanced solution, provided that the number of task instances per host is an integral value. In our case (8 hosts and 24 instances), this value is 3. This is why fixed assignment result could be taken as the upper bound for the any balanced solution.

In addition to this, the first run of the application for the balanced assignment (338 seconds) doesn't show any inferiority as compared with the other ones (341 and 343 seconds). Again, this is because of the right system assumption (all tasks having equal behavior) for balancing the tasks during their first run.

S. No.	Sequential Run	Balanced		Random		Round Robin	
	Time (sec)	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	2313	338	6.84	538	4.3	337	6.86
2	2313	341	6.78	730	3.17	346	6.68
3	2313	343	6.74	840	2.75	334	6.93
Average			6.79		3.41		6.82

Table 5.9 Completion times of CPU-Intensive application for Case A



Average Speedups

Random : RoundRobin : Balanced :: 3.41 : 6.82 : 6.79

Figure 5.10 Completion times of CPU-Intensive application for Case A

Case B: Fixed load on each host, application with variable task granularities

In Table 5.10 the random assignments (average speedup: 2.42) are still showing the worst result as in the previous case. However, now the balanced assignment clearly shows improvement over the round robin case. This is because of a considerable degree of variability involved here because of the non-similar task granularities in the application. Table 5.1 gives the granularity distribution among the application tasks. The results are sketched as a column chart in Figure 5.11.

S. No.	Sequential Run	Balanced		Random		Round Robin	
	Time (sec)	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	8209	1346	6.1	3131	2.62	1568	5.24
2	8209	1184	6.93	3617	2.27	1576	5.21
3	8209	1195	6.87	3476	2.36	1575	5.21
Average			6.63		2.42		5.22

Table 5.10 Completion times of CPU-Intensive application for Case B

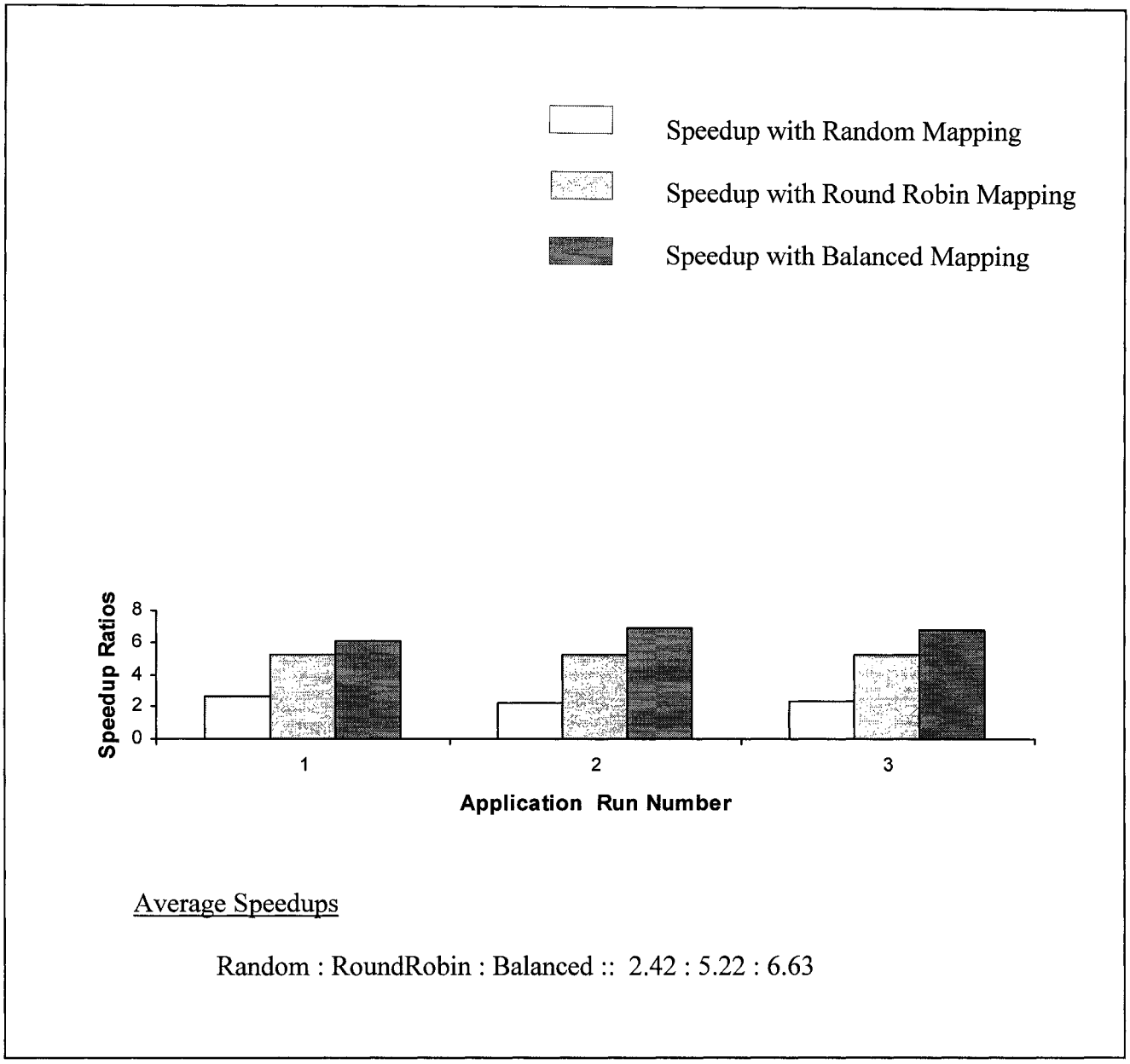


Figure 5.11 Completion times of CPU-Intensive application for Case B

Case C: Variable load on each host, application with equal task granularities

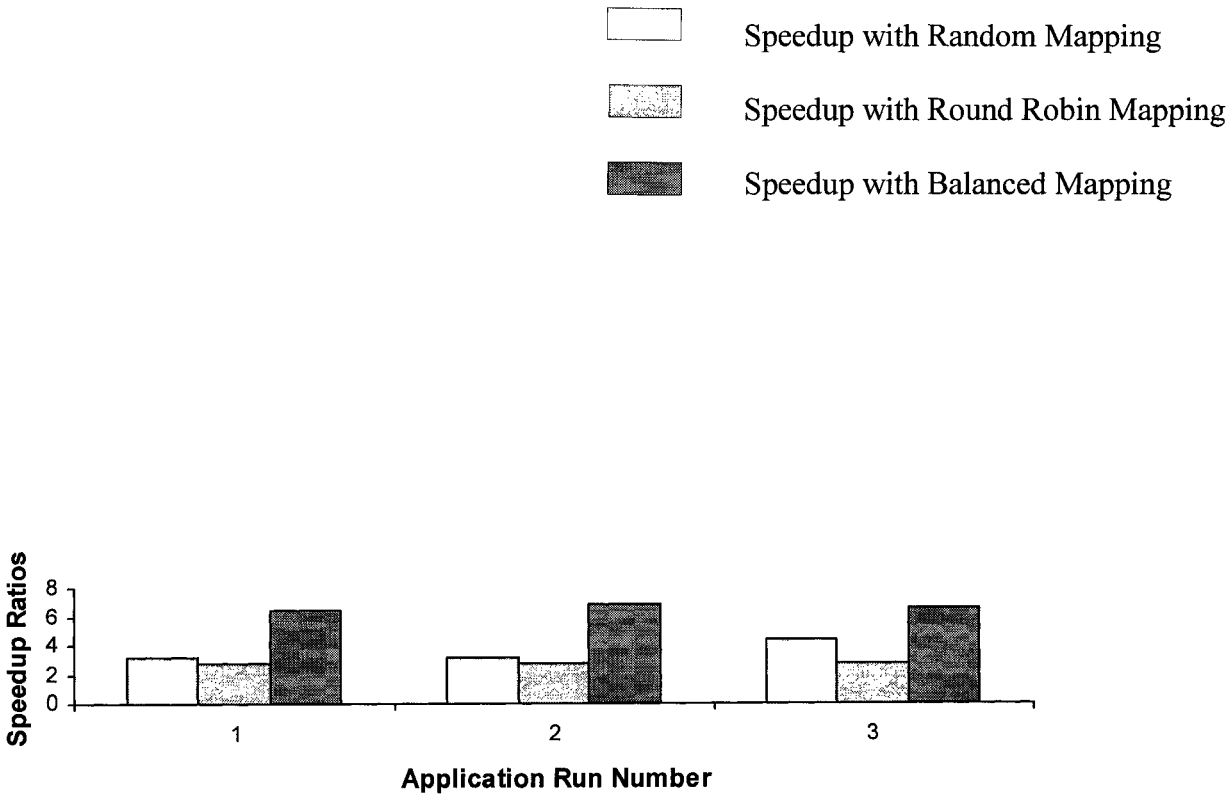
Table 5.11 shows the superiority of the balanced assignments over the other twos. The results are sketched as a column chart in Figure 5.12.

It can be seen clearly that round robin assignment has a consistent inferiority over the random results. This behavior is obvious, as round robin will always go for the same mapping regardless of the amount of imbalance exists in the environment. This explains the consistency of worst mapping in the round robin case. However, in case of random assignment, due to the existence of randomness there always exists a chance for a good assignment, although in this case random assignments are not better than balanced.

Balanced assignments are proved to be the best in all three runs of the application. Again the first run, which assumes equal task behaviors, doesn't produce much inferiority as compared with the other two, because of equal task granularities.

S. No.	Sequential Run	Balanced		Random		Round Robin	
	Time (sec)	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	2313	358	6.46	741	3.12	840	2.75
2	2313	337	6.86	728	3.18	854	2.71
3	2313	346	6.68	527	4.39	827	2.8
Average			6.67		3.56		2.75

Table 5.11 Completion times of CPU-Intensive application for Case C



Average Speedups

Random : RoundRobin : Balanced :: 3.56 : 2.57 : 6.67

Figure 5.12 Completion times of CPU-Intensive application for Case C

Case D: Variable load on each host, application with variable task granularities

Table 5.12 shows the completion times obtained for the environment of case D. The results are sketched in Figure 5.13. The values show the performance of the balanced assignment as the best. This is understandable due to the fact that the case D exhibits the worst case of imbalance as compared with the previous cases i.e. it imposes imbalance both in terms of host load and task granularities. Thus the strategies which do not consider this factor during mapping naturally exhibits inferiority in performance.

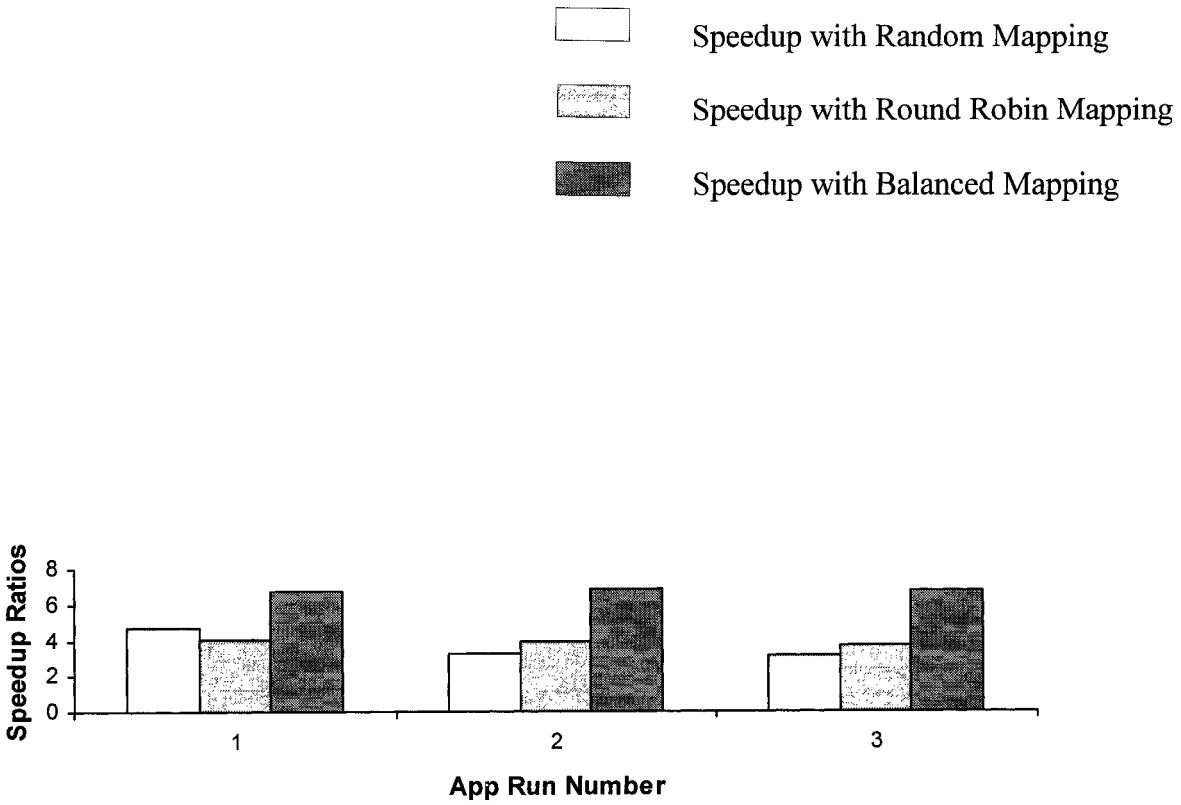
However, the performance of both random and round robin assignments is quite comparable. That is, under highly load imbalance situations, non-balanced assignments do not exhibit drastic differences among each other. This may be due to the existence of much higher time sharing overhead which mostly hides the large effects of the imbalance. This may also be the reason of having comparable results for the first application run in the balanced case (1218 seconds) with the next subsequent runs (1201 and 1217 seconds). Here the assumption of having similar granularity tasks (used in balancing the first run) was wrong. The algorithm only considers the current load on each host as the balancing criteria. Naturally some inferiority in balancing is there due to the wrong assumption but at the same time the large time sharing overheads nullify the effects of this imbalance. In the cases with none or fixed external load on system hosts, the comparatively smaller effect of time sharing could not be seen.

S. No.	Sequential Run	Balanced		Random		Round Robin	
	Time (sec)	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	8209	1218	**6.74**	1729	**4.75**	1998	**4.11**
2	8209	1201	**6.84**	2497	**3.29**	2079	**3.95**
3	8209	1217	**6.75**	2592	**3.17**	2178	**3.77**
Average			6.78		3.74		3.94

Table 5.12 Completion times of CPU-Intensive application for Case D

5.4.4.2 IO Intensive Applications

Table 5.11 to 5.14 shows the application completion times for IO intensive application obtained with different assignment strategies, for different environment cases (case A through case D) respectively. The application configuration used is the same as used in the CPU intensive case i.e. 10 tasks (each IO intensive) each having a different number of instances with the total number of instances equal to 24.



Average Speedups

Random : RoundRobin : Balanced :: 3.74 : 3.94 : 6.78

Figure 5.13 Completion times of CPU-Intensive application for Case D

Case A: Fixed load on each host, application with equal task granularities

Table 5.13 shows the completion times obtained with case A environment. The values are sketched in Figure 5.14. From the table, it is obvious that the balance strategy is outperforming the other two. However, one striking difference from the CPU intensive experiment for the same case (Case A) is that the first balanced run, shows much bigger inferiority as compared with the subsequent runs (Speedups: 5.28 Vs 7.8, 7.64).

The reason could be that the IO nature of the running tasks. Normally, in almost every operating system design, to avoid the secondary access bottleneck, all the IO is implemented through caching. The caching requires the loading of an IO block from the secondary storage at the beginning of an IO request. All the subsequent read/write will be done on the memory copy of the block. At the end, if the memory copy has experienced some change (or at fixed periods) it will be flushed to the disk by the IO routines.

Keeping the above scenario in mind, tasks in the first run needs to experience some waiting time during the loading of the disk blocks into memory. However the subsequent runs, being in the same process space of the first one (Java VM), got their targeted disk blocks already loaded in the memory. As a result these runs experience a shorter execution time. Thus, this difference is not the result of improper balancing but may happen due to an extra wait of disk activity in the first run.

The same effect could also be seen in random and round robin cases, which are showing worst time in the first run and then improvements for the others. In case of CPU intensive applications, all runs for round robin, were showing the close execution times due to the non-existence of such disk caching. The random run times are however showing random degradation due to the nature of assignment.

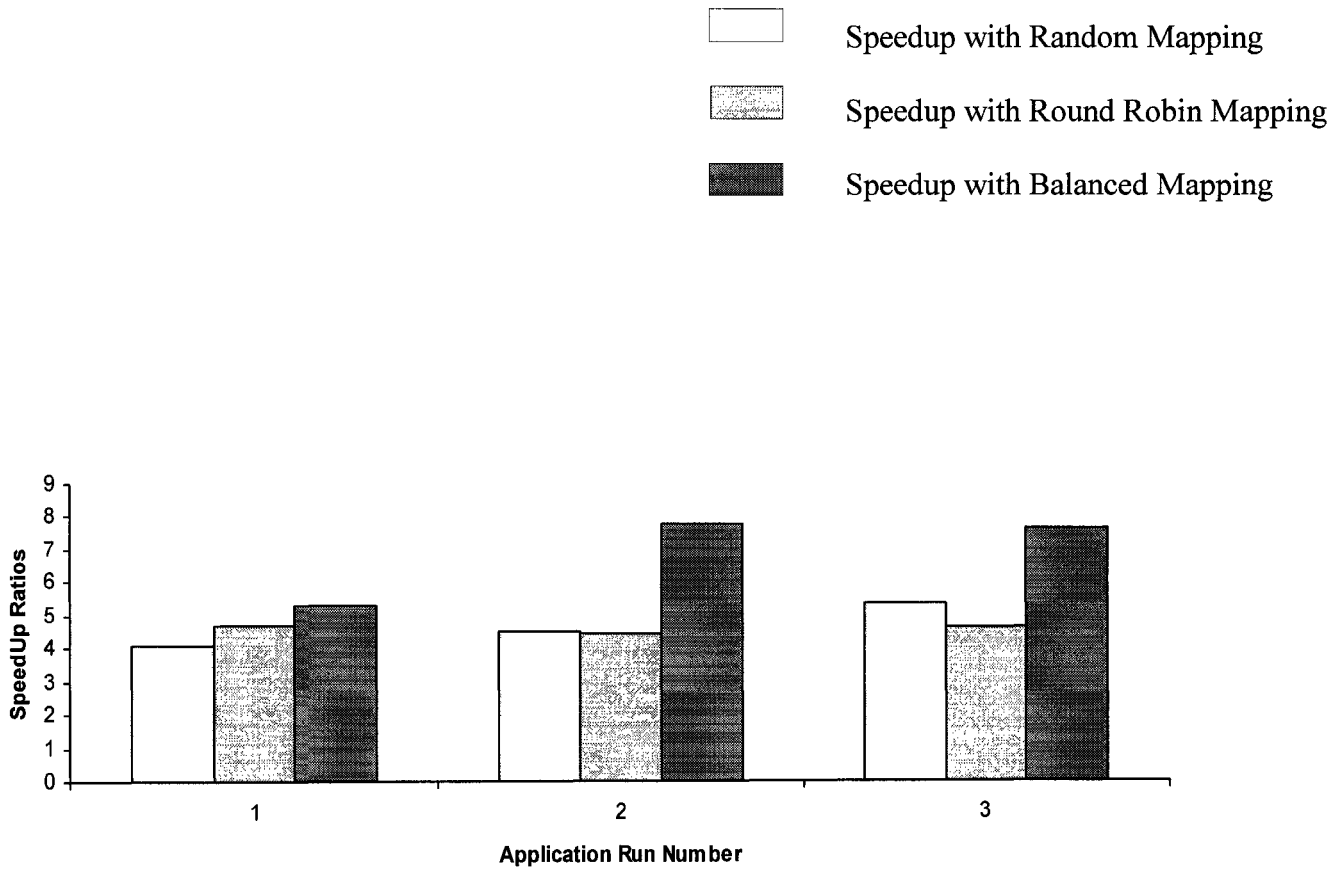
S. No.	Serial Time	Balanced		Random		Round Robin	
	Se (sec)	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	359	68	5.28	88	4.08	77	4.66
2	359	46	7.8	80	4.49	70	5.13
3	359	47	7.64	67	5.36	71	5.06
Average			6.91		4.64		4.95

Table 5.13 Completion times of IO-Intensive application for Case A

Case B: Fixed load on each host, application with variable task granularities

Table 5.14 shows the completion times and speedups obtained for IO application run for Case B. The values are plotted as column charts in Figure 5.15.

The behavior of IO disk caching is still markable in all three mappings. However due to a higher degree of imbalance involve (different task granularities), the deviation in three random runs is more than the last test. Balanced and round robin spawn values are shown to be stable, due to the determinism involved in their mappings.



Average Speedups

Random : RoundRobin : Balanced :: 4.64 : 4.95 : 6.91

Figure 5.14 Completion times of IO-Intensive application for Case A

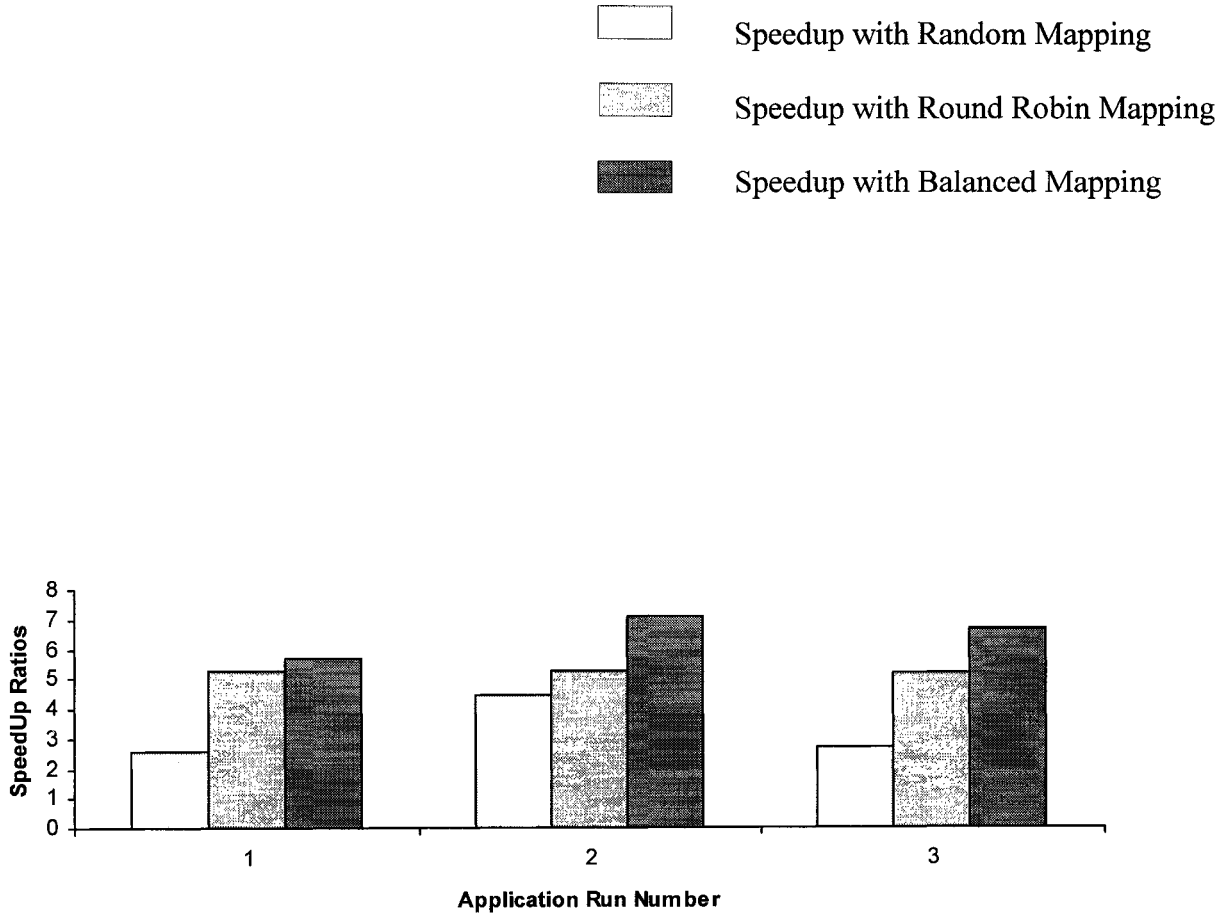
S. No.	Serial Time	Balanced		Random		Round Robin	
	Se	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	837	147	5.69	323	2.59	159	5.26
2	837	117	7.15	188	4.45	152	5.51
3	837	119	7.03	311	2.69	149	5.62
Average			6.62		3.24		5.46

Table 5.14 Completion times of IO-Intensive application for Case B

Case C: Variable load on each host, application with equal task granularities

Table 5.15 and Figure 5.16 are with the speedup results of Case-C runs. The IO caching phenomenon is still clear in all three mappings i.e. the first run in all mapping is still showing inferiority. The balanced mappings are again showing improved speedups as compared with the random and round robin distributions.

However one markedly strange behavior for the round robin assignments is that the speedups attain for IO application (average = 4.97) are more as compared with the ones achieved in CPU intensive application (average = 2.75) for the same environmental case (Case C). This shows the specialty of heavy (imbalance) load environments for the IO class of applications. This may happen due to the fact that IO task requests are handled concurrently with CPU execution. In case of CPU intensive tasks, the performance could undergo a serious degradation due to the inclusion of a larger context switching overhead, at heavy amount of loads.



Average Speedups

Random : RoundRobin : Balanced :: 3.24 : 5.46 : 6.62

Figure 5.15 Completion times of IO-Intensive application for Case B

S. No.	Serial Time	Balanced		Random		Round Robin	
	Se	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	359	64	5.61	108	3.32	74	4.85
2	359	48	7.48	72	4.99	71	5.06
3	359	50	7.18	78	4.6	72	4.99
Average			6.76		4.3		4.97

Table 5.15 Completion times of IO-Intensive application for Case C

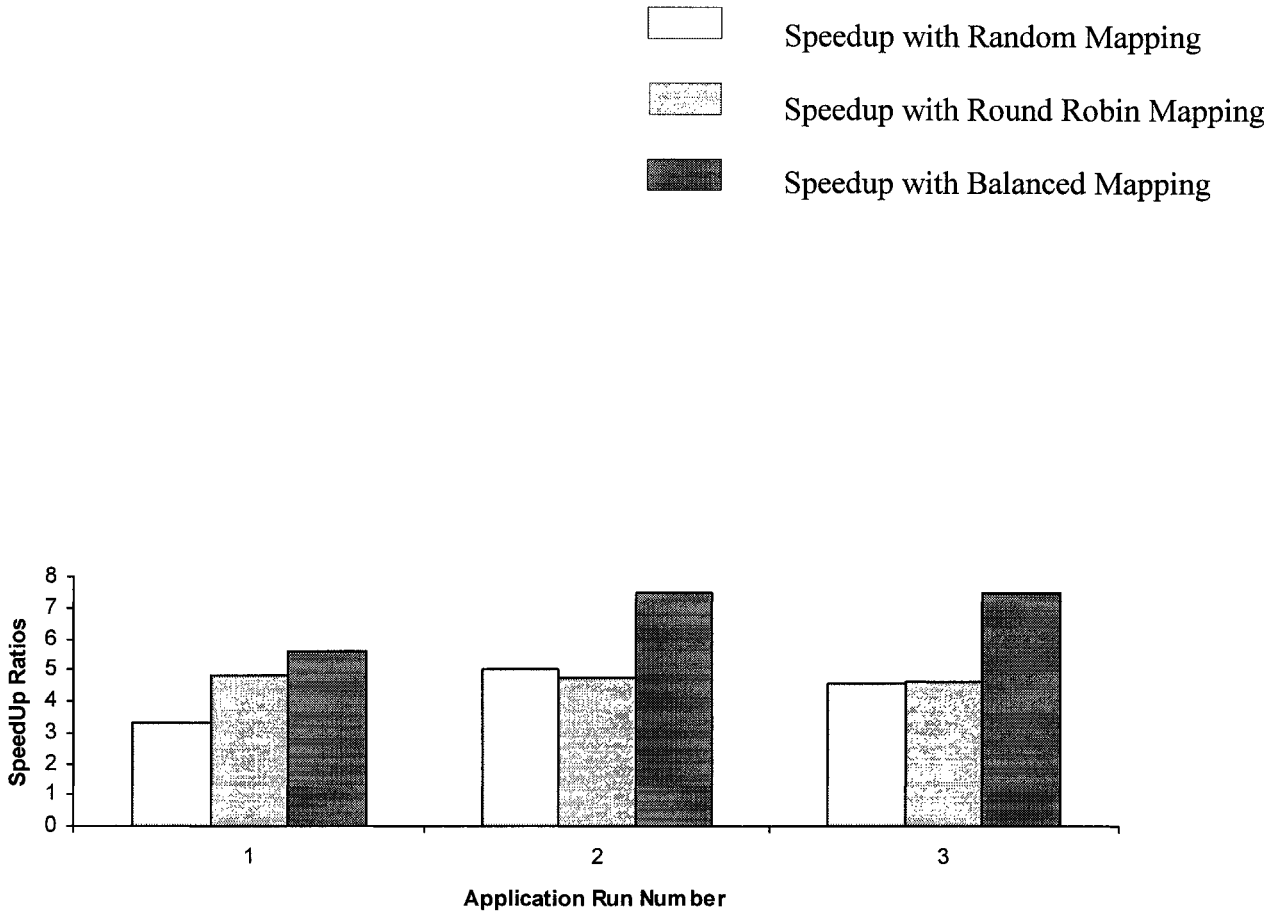
Case D: Variable load on each host, application with variable task granularities

The results for this case are shown in Table 5.16 and they are plotted in Figure 5.17.

The run behavior for each of three mappings are very similar with their corresponding values in the previous case (Case C). Again, the performance of round robing assignments (average = 5.45) are still showing much improved values over the ones in the CPU intensive (Case D runs) (average = 3.94). This further supports the claim for an IO processor working as made in the previous case (Case C).

S. No.	Serial Time	Balanced		Random		Round Robin	
	Se	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	837	164	5.1	254	3.3	159	5.26
2	837	117	7.15	268	3.12	152	5.51
3	837	121	6.92	215	3.89	150	5.58
Average			6.39		3.44		5.45

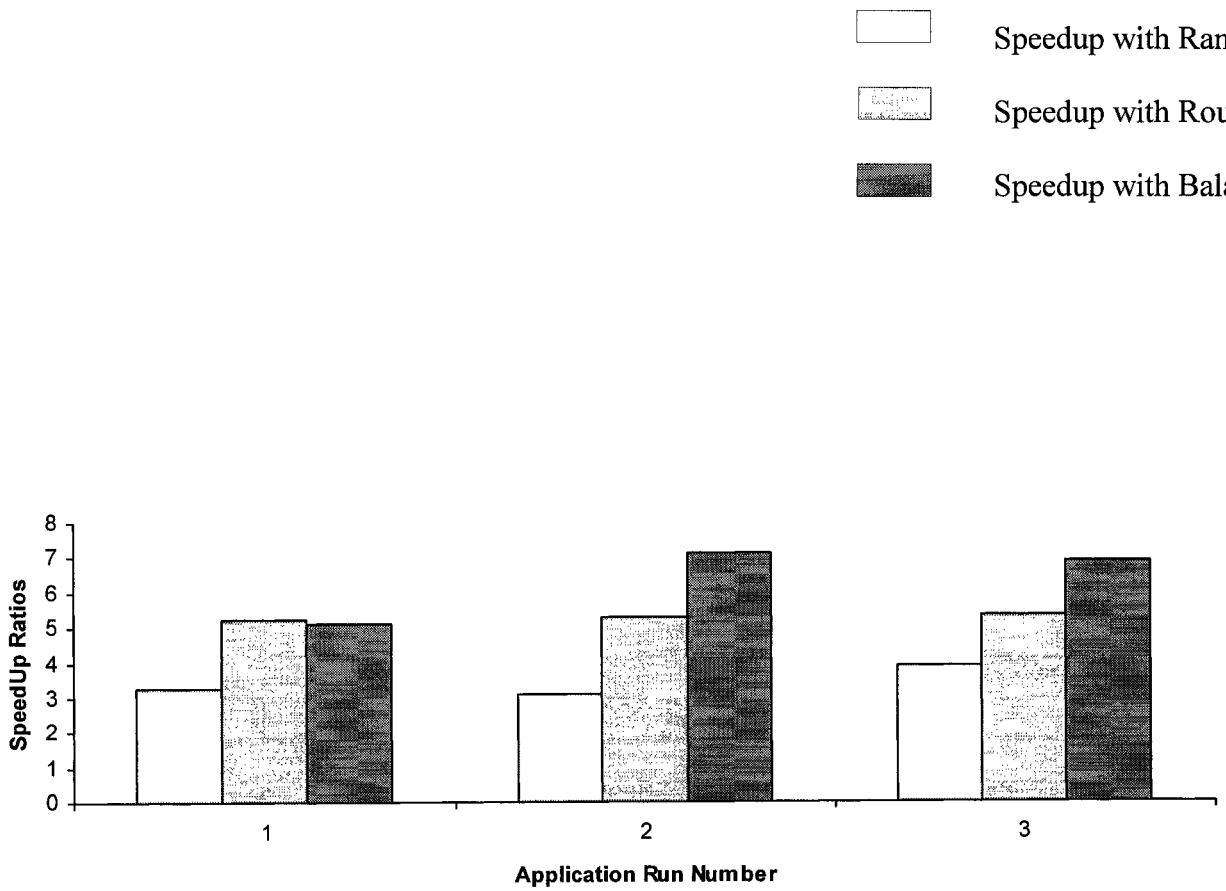
Table 5.16 Completion times of IO-Intensive application for Case D



Average Speedups

Random : RoundRobin : Balanced :: 4.3 : 4.97 : 6.76

Figure 5.16 Completion times of IO-Intensive application for Case C



Average Speedups

Random : RoundRobin : Balanced :: 3.44 : 5.45 : 6.39

Figure 5.17 Completion times of IO-Intensive application for Case D

5.4.4.3 Communication Intensive Applications

The application configuration used for communication intensive application is different from the ones used in the previous experiments (CPU and IO intensive cases). The application consists of 16 tasks, each having a single instance and having a tree like communication pattern among them.

The granularity distribution (for the application having varying task granularities) used is also different than the one used before (with CPU and IO based applications). Table 5.17 shows the granularity distribution used for the communication based tasks. For each case (A through D), single application runs are performed for each of the three heuristics: balanced, random and round robin.

Task Name	Granularity
t1	20
t2	19
t3	20
t4	18
t5	15
t6	16
t7	20
t8	18
t9	18
t10	18
t11	19
t12	15
t13	16
t14	18
t15	17
t16	23
Std. Dev.	2.09

Table 5.17 Granularity Distribution used for Communication Intensive Variable Granularity Applications

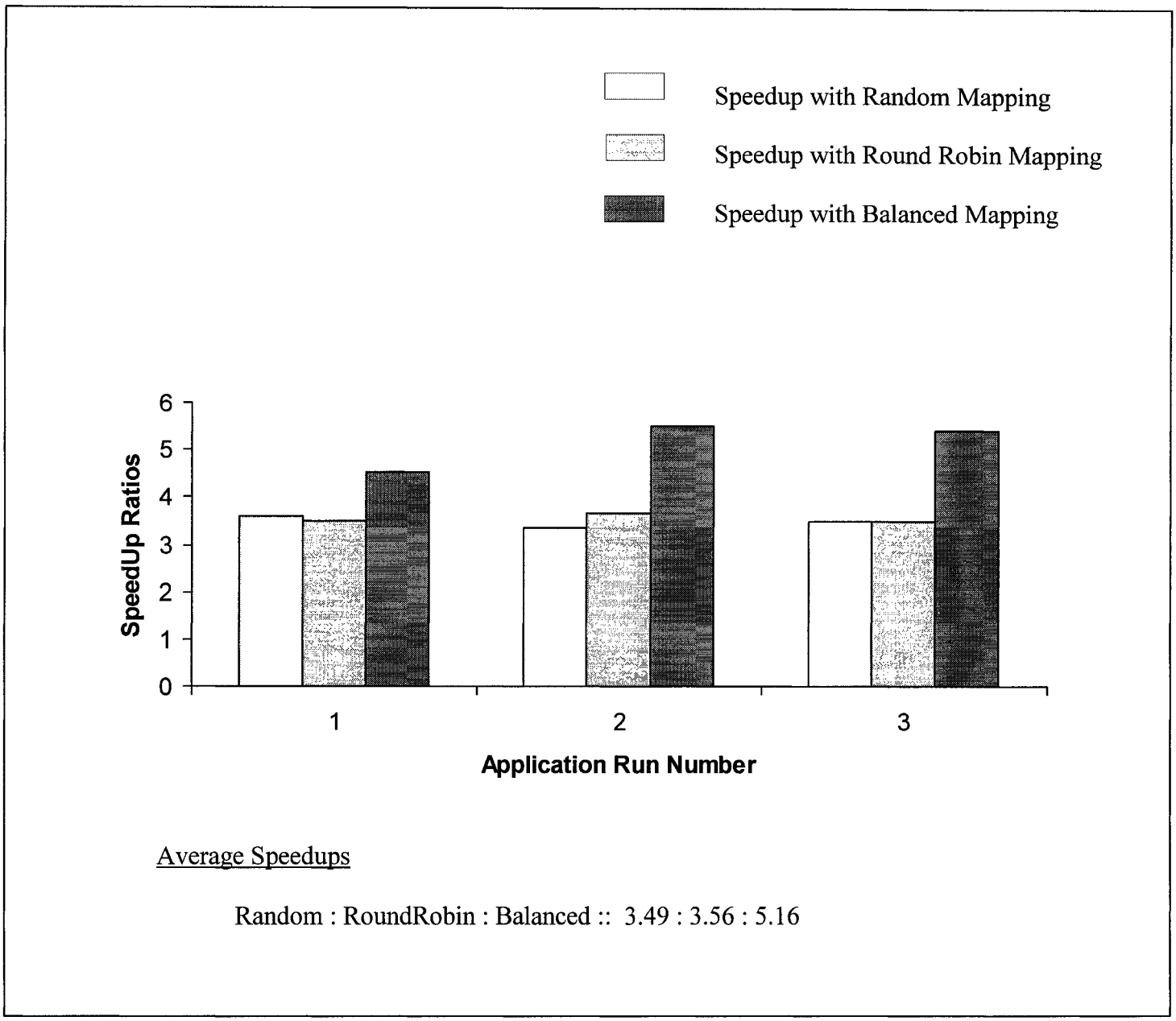
Case A: Fixed load on each host, application with equal task granularities

Table 5.18 shows the application completion times for the tree based communication intensive application with balanced, random and round robin assignment runs. The values are sketched in Figure 5.18.

The first run in case of balanced mapping showing an improved value than the runs in fixed spawns. This is because in addition with the assumption of similar task behaviors, the first run in the balanced case is making use of the application communication pattern information (supplied by the origin host as an application's task file). The round robin spawn, on the other hand, doesn't make use of any such information and does a 100% deterministic assignment (one after the other host). This is also the reason for it to exhibit similar performances for all three runs. Random assignment runs are showing closer performances with that of round robin one. This shows the heavy dependence of communication intensive application on balanced mapping, which could take care for the task localized communications. Any other mapping strategy, either deterministic (e.g. round robin) or non-deterministic (e.g. random) will eventually exhibit similar performance inferiorities.

S. No.	Serial Time	Balanced		Random		Round Robin	
	Se	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	9298	2051	4.53	2576	3.61	2651	3.51
2	9298	1687	5.51	2763	3.37	2535	3.67
3	9298	1712	5.43	2654	3.5	2665	3.49
Average			5.16		3.49		3.56

Table 5.18 Application completion times of communication-intensive application - Case A



Average Speedups

Random : RoundRobin : Balanced :: 3.49 : 3.56 : 5.16

Figure 5.18 Application completion times of communication-intensive application - Case A

Case B: Fixed load on each host, application with different task granularities

The completion time values are shown by the Table 5.19 and the Figure 5.19.

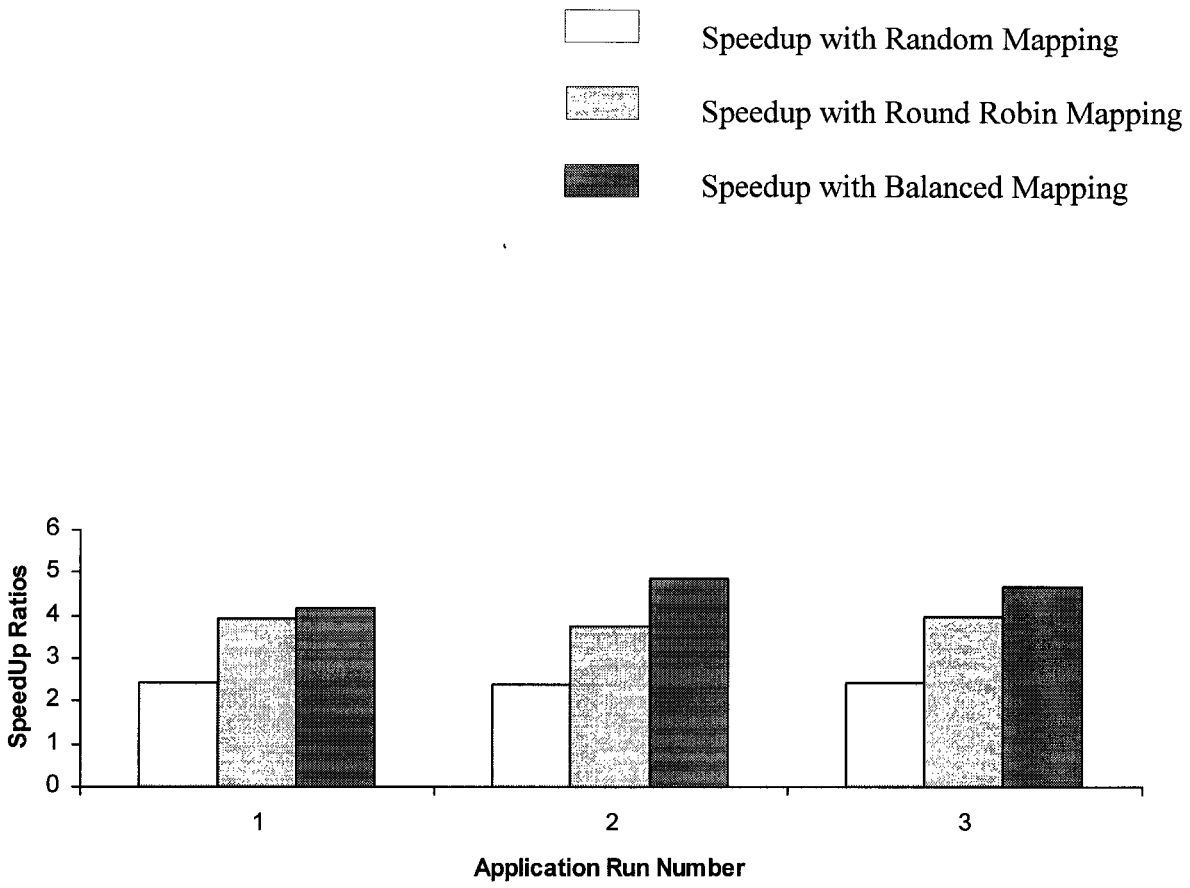
The behavior for all three mappings are similar to their respective runs in the previous case (Case A). One of the changes is a comparative lower level of speedups achieved in each case. The degradation in speedup may appear due to the increase level of time sharing overhead because of high (and variable) external load on each host. Also random assignment is now exhibiting more inferior results than the round robin which shows the superiority of a deterministic strategy over a random-based in variable load situations.

S. No.	Serial Time	Balanced		Random		Round Robin	
	Se	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	14620	3492	4.19	5943	2.46	3734	3.92
2	14620	2992	4.89	6153	2.38	3896	3.75
3	14620	3113	4.7	6043	2.42	3683	3.97
Average			4.59		2.42		3.88

Table 5.19 Application completion times of communication-intensive application - Case B

Case C: Variable load on each host, application with equal task granularities

The completion time values are shown by the Table 5.20 and the Figure 5.20. The values are exhibiting the similar behavior as in the previous case (case B). However round robin



Average Speedups

Random : RoundRobin : Balanced :: 2.42 : 3.88 : 4.59

Figure 5.19 Application completion times of communication-intensive application - Case B

assignments are showing better performance than before to prove it self as more favorable with variable external loads.

S. No.	Serial Time	Balanced		Random		Round Robin	
	Se	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	9298	2141	4.34	3026	3.07	2171	4.28
2	9298	1944	4.78	4352	2.14	1932	4.81
3	9298	1884	4.94	4254	2.19	2034	4.57
Average			4.69		2.47		4.55

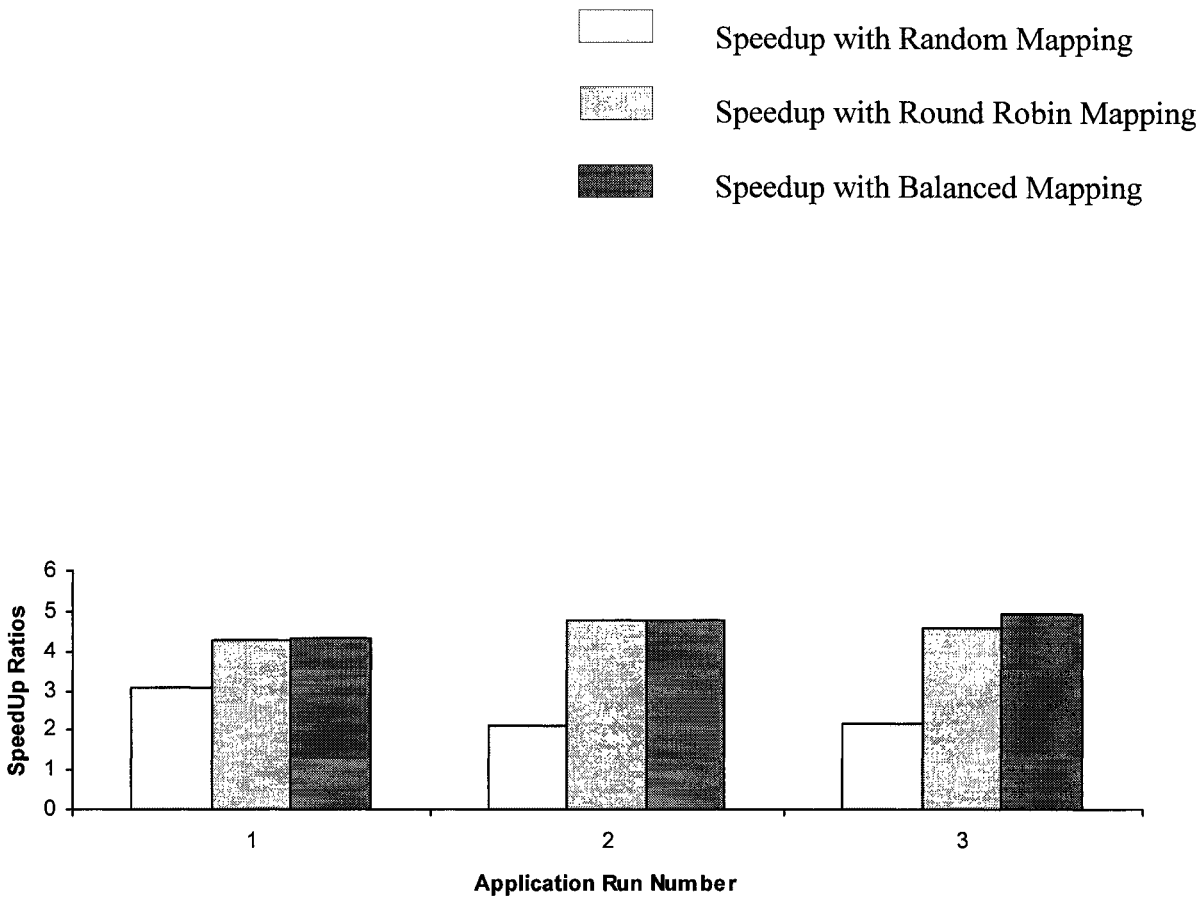
Table 5.20 Application completion times of communication-intensive application - Case C

Case D: Variable load on each host, application with different task granularities

The completion time values are shown by the Table 5.21 and the Figure 5.21. The trend among the strategies is still the similar with previous cases.

S. No.	Serial Time	Balanced		Random		Round Robin	
	Se	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	14620	3466	4.22	5029	2.91	4239	3.45
2	14620	3215	4.55	5454	2.68	4085	3.58
3	14620	3066	4.77	5543	2.64	4343	3.37
Average			4.51		2.74		3.47

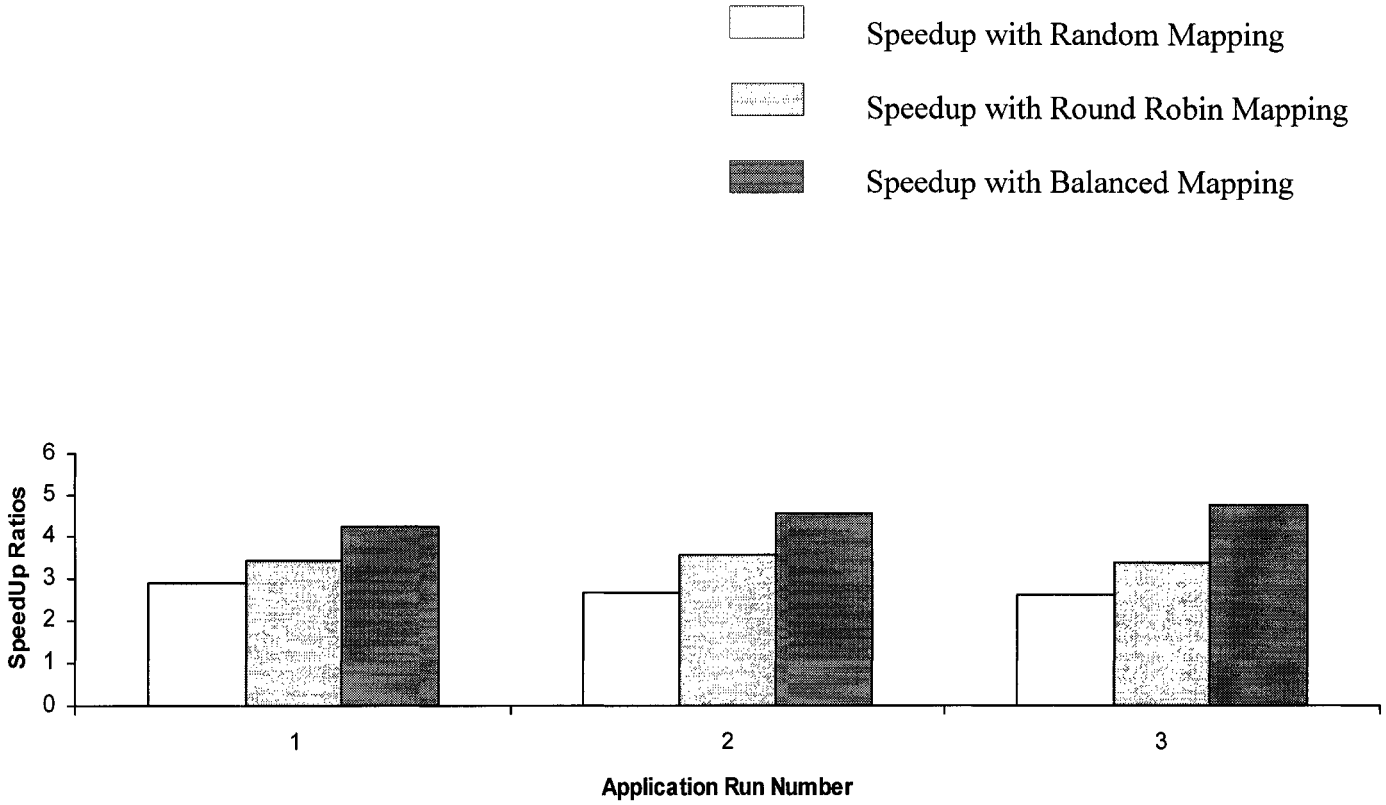
Table 5.21 Application completion times of communication-intensive application - Case D



Average Speedups

Random : RoundRobin : Balanced :: 2.47 : 4.55 : 4.69

Figure 5.20 Application completion times of communication-intensive application - Case C



Average Speedups

Random : RoundRobin : Balanced :: 2.74 : 3.47 : 4.51

Figure 5.21 Application completion times of communication-intensive application - Case D

5.4.5 Real Application - Matrix Multiplication

Matrix multiplication provides a real environment. In order to achieve good speedup levels, the operand matrices should be greater than a threshold matrix size. This ensures the application to have required level of computation and communication ratios without which, no real gain could be seen in the parallel runs. However, due to the existence of other physical limits (size of system memory etc.), there again exists a matrix size threshold at the higher extreme beyond which the speedup reduces again due to some unavoidable underlying system activities (swapping of memory pages etc.).

Thus, in the first phase of the experiment, we will use different matrix sizes and by having a keen look at their speedup values, select the size which lies in between the two thresholds. The selected size will then run both under light and heavy load conditions with the three different mapping strategies (balanced, random and round robin) to compare their performances.

5.4.5.1 Matrix Size Selection

Table 5.22 shows the speedups attained while solving matrices of different sizes in parallel. The host configuration used is 8. In order to avoid the effect of any other factor, all hosts are assumed to have fixed loads with minimum amount of network traffic on the links.

The sizes are chosen such that 24 workers[3] could be created from each of them, each of which works upon a portion of the solution space (as defined by section 5.2.2). For example, for different matrix size, the number of rows per worker, out of 24 workers, is as follows:

- 600x600 - 25 rows/worker,
- 720x720 - 30 rows/worker,
- 816x816 - 34 rows/worker,
- 912x912 - 38 rows/worker,
- 1008x1005 - 42 rows/worker, and
- 1104x1104 - 46 rows/worker

The speedup is calculated using formula S2, which requires the time for sequential and the parallel executions. Serial completion time is determined by solving the matrix on a single machine in just one loop. Parallel completion time is the time elapsed between the start and end of the matrix controller task which spawns all the other tasks, distribute the data partitions and gather results at the end.

The speedup ratios shown by Table 5.20 are considerably smaller than its ideal value (8 for 8 host configuration). The maximum of them (4.57) is just above the 50% of the ideal value. Figure 5.22 clearly shows this fact graphically. This is because of the following computation bottlenecks:

---

[3] The figure of 24 ensures the equal distribution of worker tasks among the whole 8 hosts in the system.

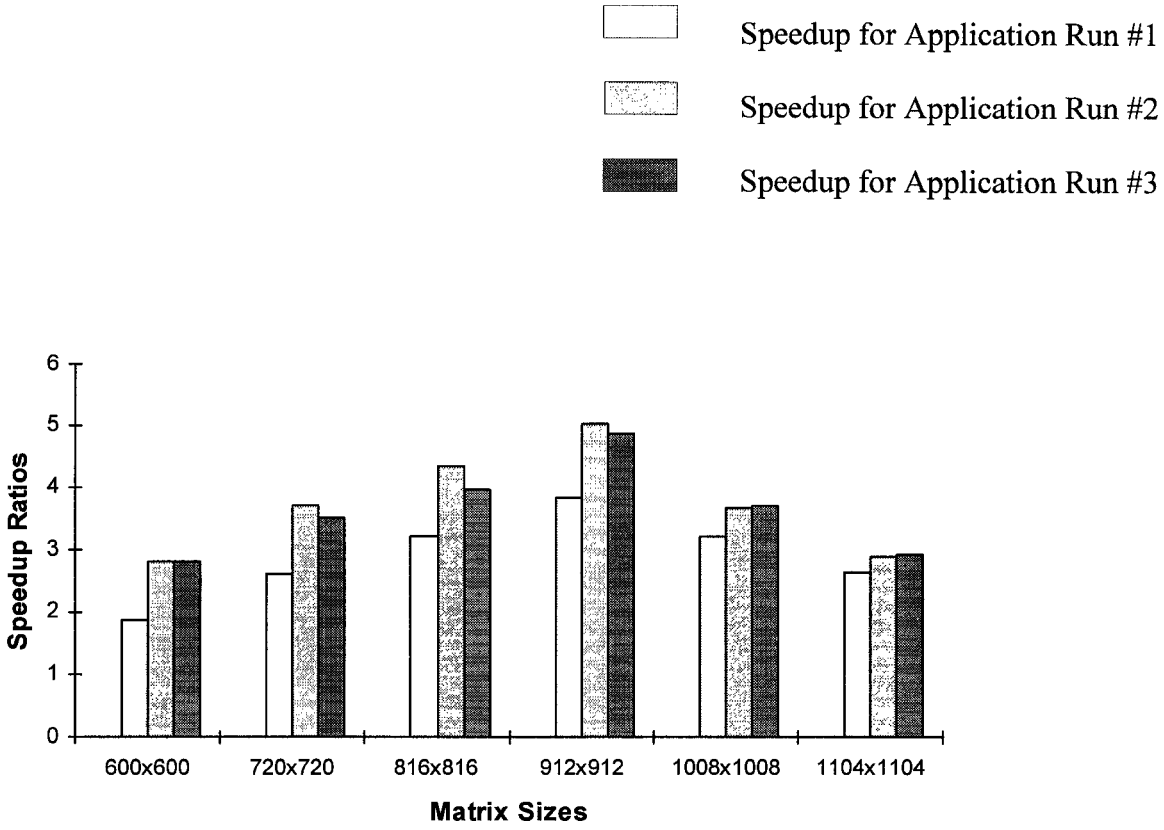
- Heavy communication involved for task interactions. For example, the controller task distributes the corresponding data partition to each of the spawned task and also received the results at the end of the computation in a sequential manner.
- Number of IO per workstation is not probably scaleable for this application. For example, every workstation must read in one of the matrices in full.

However our main concern is to choose the matrix size, which shows the biggest speedup. The table shows that 912x912 is the best of all choices because after that the speedup goes down drastically. Again the reason for such a drastic degradation could be the unavoidable physical limitations of the system memories, etc.

The test also reveals the system scalability feature in real application environments. This is implied by the growing level of speedup attain with the increase in the problem granularity up to an upper limit (912x912 in this case).

	SpeedUp (S2)					
S. No.	600x600	720x720	816x816	912x912	1008x1008	1104x1104
1	1.87	2.61	3.24	3.84	3.23	2.64
2	2.81	3.71	4.34	5.02	3.68	2.91
3	2.81	3.51	3.98	4.86	3.71	2.92
Average	2.5	3.28	3.85	4.57	3.54	2.82

Table 5.22 Speedups (S2) for different matrix sizes



Average Speedups:

600x600	2.5
720x720	3.28
816x816	3.85
912x912	4.57
1008x1008	3.54
1104x1104	2.82

Figure 5.22 Speed Ups (S2) for different Matrix sizes

5.4.5.2 Heuristic Comparison for Matrix Multiplication (912x912)

In order to compare the performance of the three mapping mechanisms for the real application, the application run is repeated with each of the three mappings: random, balanced and round robin. The effect of environment is considered by performing the tests separately for fixed and variable loads on each host, respectively.

Case A: Fixed load on each host

Table 5.23 lists the completion times and the speedup achieved by performing a parallel matrix multiplication (912x912) using three different mapping mechanisms. The results are plotted in Figure 5.23.

For all three mappings, the table consistently shows a smaller value in the first run. The reason for this is the time required at each participating host to read a copy of the matrix supplied as a local storage. For the subsequent runs, the matrix copy in the memory will be used directly.

It is also noticeable that, the speedups attained using round robin and balanced strategies are nearly the same. This is again understandable because under equal load environment, the balancing algorithm converges towards a round robin case. However, the degradation appears in the third run (as compared with the second) may happen due to the increased network traffic results because of the previous runs.

S. No.	Serial Time	Balanced		Random		Round Robin	
	Se (sec)	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	968	252	**3.84**	450	**2.15**	253	**3.83**
2	968	193	**5.02**	359	**2.7**	194	**4.99**
3	968	199	**4.86**	363	**2.67**	200	**4.84**
Average			4.57		2.51		4.55

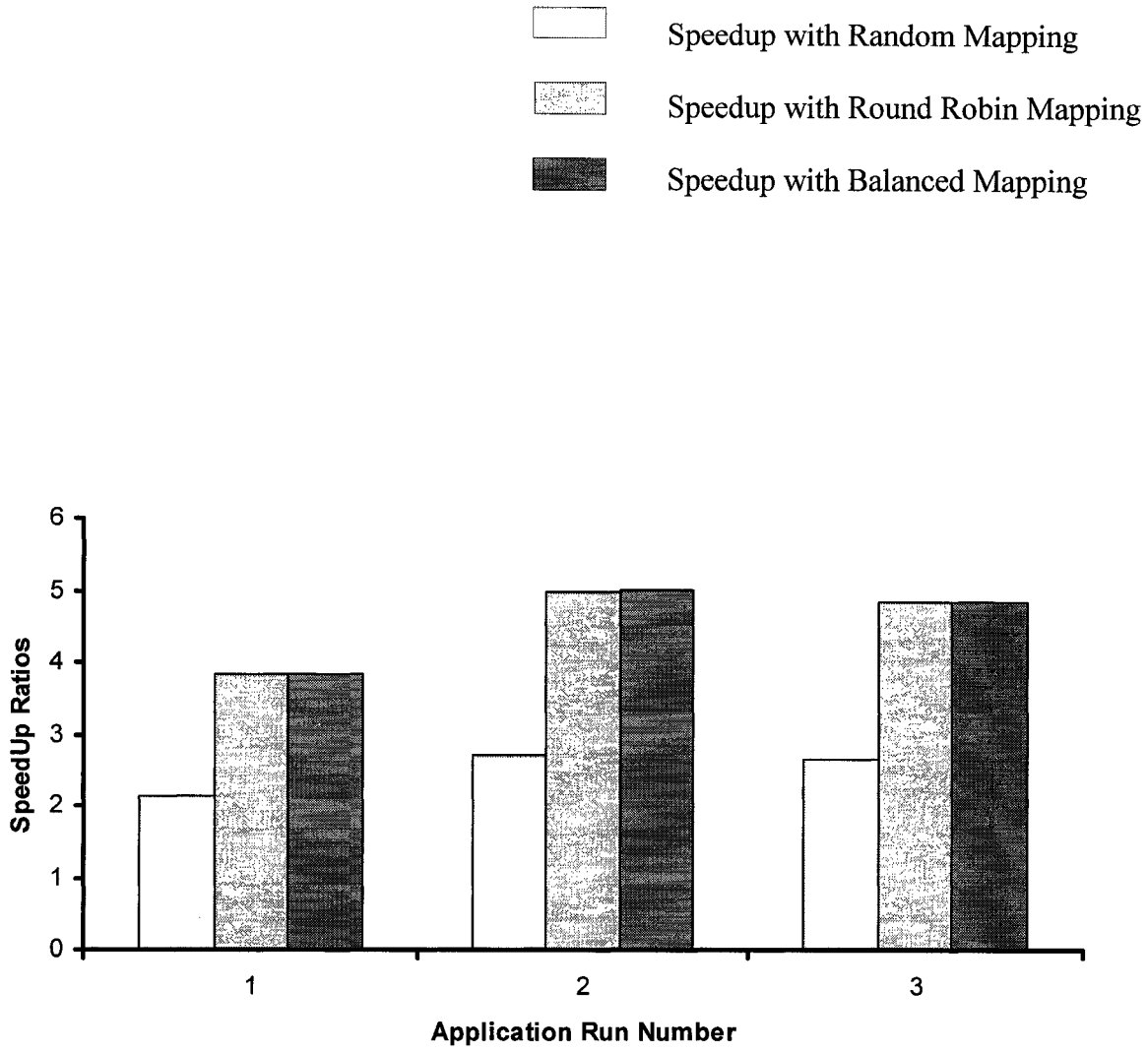
Table 5.23 Completion Times for Matrix Multiplication (912x912) for Case A

Case B: Variable load on each host

Table 5.24 lists the results for the variable load case and Figure 5.24 plots them as a column chart. Again the balanced and round robin mappings are showing similar behaviors as in the previous cases. This ensures the same performance trend of the three strategies used irrespective of the external load conditions.

S. No.	Serial Time	Balanced		Random		Round Robin	
	Se	Time (sec)	SpeedUp	Time (sec)	SpeedUp	Time (sec)	SpeedUp
1	968	261	**3.71**	465	**2.08**	257	**3.77**
2	968	274	**3.53**	365	**2.65**	268	**3.61**
3	968	267	**3.63**	376	**2.57**	270	**3.59**
Average			3.62		2.43		3.66

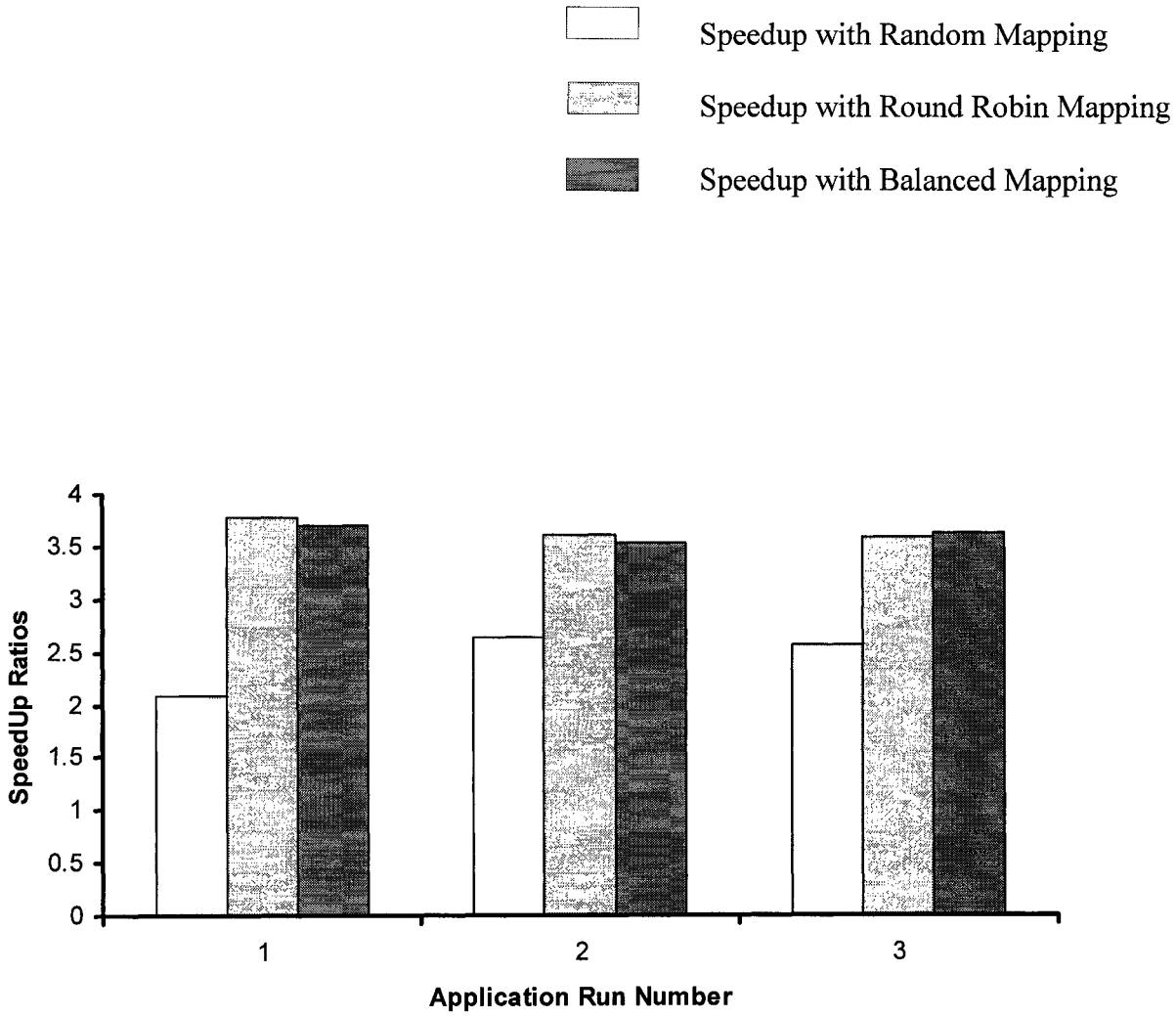
Table 5.24 Completion Times for Matrix Multiplication (912x912) for Case B



Average Speedups

Random : Round Robin : Balanced :: 2.51 : 4.55 : 4.57

Figure 5.23 Completion Times for Matrix Multiplication (912x912) - Case A



Average Speedups

Random : Round Robin : Balanced :: 2.43 : 3.66 : 3.62

Figure 5.24 Completion Times for Matrix Multiplication (912x912) - Case B

Chapter 6

Conclusion and Future Work

This research discusses the problem of load balancing for the parallel and distributed applications on a networks of heterogeneous workstations (NOW). The problem is discussed in detail on both design and implementation levels. The work has proved the viability of Java language as an underlying platform for parallel and distributed computing on NOWs under certain application conditions.

JLBS considers load balancing in a semi dynamic manner. The system load monitoring is completely dynamic, the application to NOW mapping is semi-dynamic as it is a job entry based load balancing. Maintenance of application and task history database and its use in load balancing are important features of JLBS.

Selection of Java Virtual Machine (JVM) as an implementation environment has proved the viability of JVM for such systems. The future development on JAVA are even more encouraging as far as its viability is concerned. JLBS provides support for parallel application development, system monitoring, application monitoring and load balanced application execution all in JVM environment.

The system has been tested using a generic application suit with different classes of possible applications and it is found to be scalable. The applications used involve

computation (CPU), input/output (IO) and communication (COM) intensive nature. Tests have been performed with real matrix multiplication as well. Performance is measured in terms of completion time of the applications, execution cost of tasks, and speedup achieved over sequential execution.

Experimental results have clearly shown that the performance is a function of granularity and the state of the NOW. Higher the granularity (weight of individual threads per workstation) better the speedup. The results have also shown the superiority of JLBS based task assignments over random and round robin assignments.

6.1 Future extensions

- *Comprehensive testing of different load balancing strategies:* A complete load balancing process is made up of several dominant factors. Two main factors are, for instance, load collection stage and application performance prediction. Other factors are need to be explored as well. For example, central versus distributed collection and different load metrics for the systems as well as applications etc.

- *Asynchronous Remote Method Invocation:* Although synchronous remote method invocation is supported in Java RMI package (used in current JLBS implementation), there is a need to extend it with the facility of asynchronous invocation. This will provide the programmers with the ability to overlap communication with computation. An object, which invokes an asynchronous

method, is then free to continue execution until a point at which a returned value is needed (if any).

- *Theoretical performance analysis of the developed framework*: The performance of JLBS can be verified by a mathematical model, to strengthen its theoretical basis.

- *A dynamic process migration subsystem:* Process migration requires record of low level run time task state. The current implementation of JLBS does not interfere with this aspect. It is based on an initial task placement strategy. However the support for active task migration already exists in Java, named Java Object Serialization. This facility could be utilized for implementing a process migration subsystem providing check-pointing, roll back and restarting of running tasks. Such subsystem, if incorporated into JLBS, would allow adapting to dynamic load changes throughout the network, thus making it fully dynamic.

- *Support for powerful development tools:* Powerful programming environments are vital for the popularization of parallel applications. A parallel compiler and debuggers are yet to be available to shorten the application development cycle.

- *Purely Distributed JLBS:* In the current implementation, a hybrid approach for making balancing decisions is employed. Balancing is done within a host domain boundary. However this could be made purely distributed in which each

application agent makes the mapping decision locally. Obviously, this happens at the expense of higher cooperation overhead. On the other hand, the pure distributed nature could result in much higher scalability than exhibited by the current implementation. However, the feasibility of this implementation mainly depends upon the underlying communication subsystem. It will be feasible in case communication is not a bottleneck under certain application conditions.

- *Incorporating fault tolerance:* At system level, this could involve tolerating faults in any of the JLBS system objects such as the application agent, load agent, application controller, load controller etc. At application level, check-pointing and migrations can provide higher level fault tolerance.

References

[1] M. Bozyigit, K. Al. Tawil and S.K. Naseer, "A task migration facility for parallel & distributed applications", *PDPTA International Conference,1997,* pp. 1699-1704, 1997

[2] B. Christiansen, P. Cappello, M.F. Ionescu, M.O. Neary, K. Schanuser. And D. Wu. "Javelin: Internet based parallel computing using Java", *Concurrency: Practice and Experience,* Vol. 9, No. 11, pp. 1139-1160, Nov. 1997

[3] S. Nisar ul Haq, M. Bozyigit, S. Ghanta, and S. K. Naseer, "Design of a Load Balancing Framework for Distributed and Parallel Applications", *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA-95),* Vol. 1, pp. 845-854, 1995

[4] Cap, C. H., and Stumpen V., "Efficient parallel computing in distributed workstation environments", *Parallel Computing* Vol. 19, pp. 1221-1234, 1993

[5] M. Cermele, M. Colajanni, G. Necci, "Dynamic Load Balancing of Distributed SPMD Computations with Explicit Message-Passing", *Proceedings, Sixth Heterogeneous Workshop (HCW97),* pp 2-16, 1997

[6] C. –Z. Xu and F. C. M. Lau. "Decentralized remapping of data-parallel computations with the generalized dimension exchange method",. *In Proceedings of 1994 Scalable High Performance Computing Conference*, pp. 414-421, May 1994

[7] C. Kim and H. Kameda, "An algorithm for optimal static load balancing in distributed computer systems", *IEEE Trans. on Compute.,* Vol. 41, No. 3 pp. 381-384, March 1992

[8] Yan Gu, B.S. Lee, Wentong Cai, "Evaluation of Java Thread Performance on Two Different Multithreaded Kernels", *Operating Systems Review,* Vol. 33, No. 1, pp. 34-46, Jan 1999

[9] D. M. Nicol and Saltz J. H., "Dynamic re-mapping of parallel computations with varying resource demands", *IEEE Transactions on Computers*, Vol. 37, No. 9, pp. 1037-1087, September 1988

[10] Dingxing WANG, Weimin ZHENG, Jianxin XIONG, "Research on Cluster of Workstations", *I-SPAN '97, Proceedings, Third International Symposium on Parallel Architectures, Algorithms & Networks,* pp. 275-281, 1997

[11] F. C. H. Lin and R. M. Keller,"The gradient model load balancing method", *IEEE Transactions on Software Engineering,* Vol. 13, No. 1, pp. 32-38, January 1987

[12] Geist, A. Beguelin, A. Dongarra, J. Jiang, W. Manchek, R. and Sunderam, PVM: Parallel Virtual Machine - A User's Guide and tutorial for Networked Parallel Computuing, *MIT Press*, 1993

[13] M. Ghouse-uddin, M. Bozyigit, "A virtual distributed computing system", M.S. Thesis, Computer Science Department, KUPFM, 1998

[14] G. Horton, "A multi-level diffusion method for dynamic load balancing", *Parallel Computing,* Vol. 19, No. 1, pp. 209-218, 1993

[15] Gilbert Cabillic and Isabelle Puaut, "Stardust: An Environment for Parallel Programming on Networks of Heterogeneous Workstations", *Journal of Parallel and Distributed Computing,* Vol. 40, pp. 65-80, 1997

[16] K. K. Goswami, M. Devarakonda, and R. K. Iyer, "Prediction based dynamic load-sharing heuristics", *IEEE Trans. Para. Dist. System,,* Vol. 4, No. 6 pp. 638-648, June 1993

[17] J. Gosling, B. Joy and G. Steel. *The Java Language Specification,* Addison-Wesely, 1996

[18] O. Larson, M. Feig and L. Johnsson, "Sommet computing experiences for scientific applications", *P. Processing Letters,* Vol. 9 No.2, pp. 243-252, 1999

[19] L. M. Ni, C. –W. Xu, and T. B. Gendreau, "A distributed drafting algorithm for load balancing", *IEEE Transactions on Software Engineering*, Vol. 11, No.10, pp. 1153-1161, October 1985

[20] H.-C.Lin and C.S. Raghavendra, "A dynamic load balancing policy with a central job dispatcher (LBC)", *In Proceedings of 11th* International conference on Distributed Computing Systems, pp. 264-271, February 1992

[21] M. Izzat, Patrick Chan and Tim Brecht. "Ajents: Towards an Environment for Parallel, Distributed and Mobile java Applications", *In ACM 1999 Java Grande Conference*, 1999

[22] Mounir Hamdi, Yi Pan, B. Hamidzadeh, and F.M. Lim, "Potentials and Limitations of Parallel Computing on a Cluster of Workstations", *Proceedings 1997 International Conference on Parallel & Dist. Systems* pp. 572-577, 1997

[23] Michal Cierniak, M. Javed Zaki and Wei Li, "Compile Time Scheduling Algorithms for a Heterogeneous Network of Workstations", *The Computer Journal,* Vol. 40, No. 6, pp. 356-372, July 1997

[24] N. G. Shivaratri and P. Krueger. "Two adaptive location policies for global scheduling algorithms", *In Proc. 10th Int. Conf. Dist. Computing Systems,* pages 502-509, May 1990

[25] P. Mehra and B. W. Wah. "Load Balancing: An Automated Learning Approach", *World Scientific Publishing Co. Pve. Ltd.* 1995

[26] R. Mirchandaney, D. Towsley, and J. A. Stankovic, "Adaptive load sharing in heterogeneous distributed systems." *Parallel and Distributed Computing* Vol 9, pp. 331-346, 1990

[27] R. Bisani and A. Forin, "Multilanguage Parallel Programming of Heterogeneous Machines", *IEEE Trans. Computers,* Vol. 37, No. 8: pp. 930-945, Aug 1988

[28] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends", *Computer,* Vol. 19, No. 8 pp. 26-34, Aug 1986

[29] Ted G. Lewis, "Foundations of Parallel Programming: A machine independent approach. *IEEE Computer Society Press*, 1993

[30] T.F. Znati, R. G. Melhem, and K. R. Pruhs. "Dilation-based bidding schemes for dynamic load balancing on distributed processing systems", *In Proceedings of 6th Dist. Memory Comp. Conf.* , pp. 129-136, April 1991

[31] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing", *Journal of Concurrency: Practice and Experience,* Vol. 2, No. 4, pp. 315-339, December 1990

[32] J. Worlton, "Towards a Science of Parallel Computation, Research Monographs in Parallel and Distributed Computing". New York, 1986

[33] Wolfgang Obeloer, Claus Grewe, Holger Pals, "Load Management with Mobile Agents", *Proceedings of 24th Eoromicro Conference 1998,* Vol. 2, pp. 1005-1012, 1998

[34] Y. Zhang, H. Kameda, and K. Shimizu, "Parametric analysis of optimal static load balancing in distributed computer systems", *J. Inf. Process,.* Vol 14, No. 4, pp. 433-441, 1991

[35] Zhou, S, Stumm, M. Li. K. and Worman, "Heterogeneous distributed shared memory", *IEEE Transaction on Parallel Distributed Systems* Vol. 3 No. 5 pp. 50-54, Sept 1992

[36] Ahmed, A. Ghafoor, and G. Fox. "Hierarchical scheduling of dynamic parallel computations on hypercube multi-computers", *Journal of Parallel and Distributed Computing*, Vol.17, No. 8, pp. 317-329, March 1994

[37] A Alexandrov, M. Ibel, K. Schauser, and C. Scheiman, "Super Web: Towards a global Web based parallel computing infrastructure", In 11th International Parallel Processing Symposium, pp. 100-106, April 1997

[38] Arash Baratloo, Mehmet Karaul, Holger Karl, and Zvi M. Kedem, "An Intrastructure for Network Computing with Java Applets", In *ACM 1998 Java Grande Conference,* 1999

[39] A Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte, "Meta computing on the Web", In *Proceedings of the 9th International Conference on Parallel and Distributed Computing Systems,* 1996

[40] OMG. CORBA/IIOP *2.2 specification,* June 1998

[41] G. Fox and W. Furmanski, "High Performance Commodity Computing", In I. Foster and C. Kesselman, editor, *The GRID. Blueprint for a New Computing Infrastructure,* Morgan Kaufmann, 1999

[42] C.G. Harrison, D.M. Chess, A. Kershenbaum, "Mobile Agents: Are they a Good Idea?", *Research Report,* IBM Res. Div., T.J. Watson Center, 1995

[43] IMPI Steering Committee. "IMPI - Interoperable Message-Passing Interface, DRAFT", National Institute of Standards and Technology, August 1998

[44] Jerrel Watts and Stephen Taylor, "A practical approach to dynamic load balancing", *IEEE Trans. Par. Dist. Systems* Vol. 9, No. 3, pp. 235-248, March 1996

[45] Jerrel Watts, Stephen Taylor and Sirkunya Nilpanich, "SCPlib: A Concurrent Programming Library for Programming Heterogeneous Networks of Computers", *Information Technology Conference, IEEE,* pp 153-156 1998

[46] Kranz, D. Johnson, K. Agarwal, A. Kubiatowicz, J. and Lim, B.-H. "Integrating message-passing and shared-memory: Early experience." *Proc. of the Fourth SIPLAN*

Symposium on Principles and Practice of Parallel Programming, pp 54-63, 1993

[47] K.Kumar, and N. Ichiyoushi "Probabilistic analysis of the optimal efficiency of the multi-level dynamic load balancing schemes". *In Proceedings of 6th Distributed memory computing Confierence*, pp. 145-152, April 1991

[48] Govind Seshadri, "Distributed Java Computing Using RMI", *Java Report,* Vol. 2, No. 10, pp. 39-46, November 97

[49] K. G. Shin and Y. –C. Chang. "Load sharing in distributed real-time systems with state-change broadcasts". *IEEE Transactions on Computers*, pp. 1124-1142, August 1989

[50] M. Litzkow, C. Livny, and M. Mutka, "Condor-A hunter of idle workstations". *Proc. of 8th Int'l Conf. on Distributed Computing Systems, pp. 104-111* June 1988

[51] L. Kipp(ed). *Perfect Benchmarks Doc, Research Report, SUITE 1,* CSRD, Univ. of Illinois, Urbana-Champaign, Oct 1993

[52] D. S. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, S. Zhou, "Process Migration Survey", *Collected Papers (Research Report),* Vol. 5, The Open Group Research Institute, March 1997

[53] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 1.1*, Research Report, June 1995

[54] Message Passing Interface Forum. *MPI-2: Extension to the Message-Passing Interface Standard*, Research Report, July 1997

[55] D. P. Bertskas and R. Gallager, "Computer Communication Networks.", *Prentice Hall: Englewood Cliffs* , 1989

[56] G. J. Miller, K. Thompson, and R. Wilder, "Performance Measurement on the vBNS,",

In Proceedings of the Interop '98 Engineering Conference, Las Vegas, NV, May 1998

[57] ObjectSpace, "ObjectSpace Voyager Core Package Version 1.0", *Technical Overview,* ObjectSpace, Inc. 1997

[58] Mohammed Javeed Zaki, Wei Li, Srinivasan Parthasarathy, "Customized Dynamic Load Blancing for a Netowork of Workstations", *Proceeding of 5th IEEE International Symposium on High Perf. Dist. Computing, pp. 282-291,.* 1996

[59] A. Weinrib aand S. Shenker, "Adaptive load sharing in large heterogeneous systems", *In Proceedings of the IEEE INFO COM,* 1988

[60] Yongbing Zhang, Katsuya Hakozaki, Hisao Kameda and Kentaro Shimizu, "A Performance Comparison of Adaptive and Static Load Bal. in Heterogenous Dist. Sys.", *Proc. of the 28th Annual Simulation Symposium,* pp. 332-340, 1995

[61] Alex McManus and John Hunt, "The Need for Speed", *Java Report* ,Vol. 3, No. 5, pp. 39-44, May 1998

[62] Java 1.2 Application Programming Interface (API) Specifications, *Sun Microsystems*, 1999

Vita

Name	Irfan Ahmed Ilyas
Date of Birth	1st July, 1971
Nationality	Pakistani
Bachelors Degree	Bachelor of Engineering in Computer Systems (December 1995) from NED University of Engineering and Technology, Karachi Pakistan
Masters Degree	Master of Science in Computer Science (April 2000) from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia