AN ABSTRACT OF THE THESIS OF

Joseph Jacob for the degree of Master of Science in Computer Science presented on April 4th, 1995. Title:

Automatic Scheduling and Dynamic Load Sharing of Parallel Computations on Heterogeneous Workstation Clusters

Abstract approved: _____

Vikram Saletore

Parallel computing on heterogeneous workstation clusters has proved to be a very efficient use of available resources, increasing their overall utilization. However, for it to be a viable alternative to expensive, dedicated parallel machines, a number of key issues need to be resolved. One of the major challenges of heterogeneous computing is coping with the inherent heterogeneity of the system, with the availability of workstations from different vendors of varying processing speeds and capabilities. The existence of multiple jobs and users further complicates the task. The time taken for a parallel job is constrained by the time taken by the slowest or the most heavily loaded workstation. Therefore, load sharing of parallel computations is imperative in ensuring good overall utilization of the system. Since load sharing is essentially independent of the particular parallel job being run, the development of program independent, automatic, scheduling and load sharing strategies have become vital to the efficient use of the heterogeneous cluster. This thesis discusses various prior approaches to load sharing, examines a new strategy developed for heterogeneous workstations, and evaluates its performance.

Automatic Scheduling and Dynamic Load Sharing of Parallel Computations on Heterogeneous Workstation Clusters

by

Joseph Jacob

A Thesis

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed April 4th, 1995
Commencement June 1996

Master of Science thesis of Joseph Jacob presented on April 4th, 1995

APPROVED:

Redacted for Privacy

Major Professor, representing Computer Science

Redacted for Privacy

Head of Department of Computer Science

Redacted for Privacy

Dean of Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Redacted for Privacy

Joseph Jacob, Author

ACKNOWLEDGMENTS

This thesis would have remained largely incomplete if it weren't for long nights, Manojith Padala and caffeine, not necessarily in that order. Sometimes I wonder what this thesis would have looked like without their help. I would like to take this opportunity to thank Dr Vikram A. Saletore for his continued support, ideas and patience over the last two years. Dr Michael J. Quinn deserves a special mention of thanks for sharing his vast insight and grasp of parallel processing with me. I have deep appreciation and gratitude to Dr Walter G. Rudd for agreeing to be on my committee and being always available to listen to my ideas and suggestions, despite his busy schedule. Dr Roger C. Graham has my appreciation for cheerfully agreeing to be my Graduate School Representative at short notice. Vinod Sharma has to be acknowledged for his critiques, bringing clarity and a sense of purpose to my writing. Any digressions and flagrant violations of the English language that remain are mine and mine alone. I deeply appreciate the help and support of all my friends. Without their understanding and friendship, I could never have finished this work. Finally, my fondest regards and gratitude to my parents and my sisters, Raji and Ria. Their constant unwavering support and faith in me, despite all the trials and tribulations, continually amazes me. I hope this thesis remains a testament to their support and faith.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

DEDICATION

For those of you, who continued to have faith in me

-even when I had lost mine

AUTOMATIC SCHEDULING AND DYNAMIC LOAD SHARING OF PARALLEL COMPUTATIONS ON HETEROGENEOUS WORKSTATION CLUSTERS

1. INTRODUCTION

1.1. BACKGROUND

Parallel computing on workstation clusters has become a very viable option for optimal use of available resources. Workstations connected by an interconnection network can be considered to be a single computing entity that can handle computation intensive tasks that would otherwise take too long on a particular workstation. Extremely powerful processors like the Digital Alpha, UltraSPARC, Intel P6 and high bandwidth, low latency networks have ensured that parallel computing on workstation clusters is an alternative to expensive, dedicated, high performance parallel machines. Educational institutions and the industry are increasingly unwilling to make a huge investment and commitment on a particular parallel machine or parallel architecture in the absence of any consensus on an industry standard or the best architecture.

Workstation clusters are popular for a number of reasons. Typically, because of the small client base, new parallel machines with the latest processors take more time to be available than workstations using the very same processors. This lag and the extremely rapid rate at which processor speeds have doubled over the last

decade - roughly once every eighteen months - have resulted in parallel machines that have consistently smaller MFLOPS ratings per processor than the processors used in high-end workstations. The complex nature of the underlying operating system and specialized hardware has made parallel machines more prone to failure than workstations. Upgrading parallel machines is also neither easy nor inexpensive. Workstations can be added to a existing network within hours. They also allow individual users to do word processing, send and receive e-mail and run other applications in addition to performing parallel computations. Since a larger volume of workstations are sold per year in comparison to dedicated parallel machines, the performance to price ratio is very favorable for workstations. When users run interactive applications, a number of CPU cycles are wasted which could possibly have been used for some useful purpose. This availability of free CPU cycles as well as the better performance to price ratio could be the best motivating factors for the rapid growth of parallel computing on workstation clusters.

Programming on workstation clusters introduces a number of problems that never existed with parallel machines. The primary stumbling block is the absence of environments that provide true operating system and vendor independence. This is needed to harness all the available resources as educational institutions and the industry typically have workstations from different vendors. A number of machines use different forms of data representations and alignment characteristics. This necessitates the use of using a machine independent form of data representation like XDR or ISIS which have their associated overheads. The available memory and CPU speed of the workstations also vary widely among themselves. Machines can be connected by FDDI, ATM, Ethernet or some other local area network. Debugging and development of applications on workstation clusters is very difficult because of the

absence of convenient tools. Each machine in a typical network is not available for dedicated parallel use. This means at different instants, the workload on individual workstations will vary. To efficiently use such a cluster, it is imperative that we use an adaptive load balancing scheme that reduces the total turnaround times for parallel jobs by moving work from heavily loaded processors to more lightly loaded ones.

A number of libraries and environments have been developed to provide some level of support to heterogeneous computing. PVM, P4, Mentat, Charm, Dataparallel C and Condor are some examples. Some low level debugging tools like Xab exist but very few comprehensive debugging tools are available for heterogeneous clusters. Several of the languages and environments support the data parallel paradigm while others follow the control parallel paradigm.

1.2. RELATED WORK

Despite the plethora of environments and libraries available, the amount of related work is not significant. Libraries like PVM [8], P4, TCGMSG are low level message passing libraries and provide very little run-time support. Sprite [3], Condor [14] and Stealth [12] look at problems of extremely large data sizes with little or no communication. Transfer of data in such environments usually implies a process migration. This necessitates a complicated method of checkpointing and restart or redundant process execution on multiple workstations. The class of problems this report looks into is what is referred to as medium grain-sized parallel problems and run-time environments that provide some degree of support, particularly automatic scheduling and dynamic load balancing.

1.2.1. Mentat

Mentat is a dynamic, object oriented parallel programming system developed at the University of Virginia [9]. Parallel programs are written in an object oriented language called the Mentat Programming Language (MPL) which can then be run on heterogeneous workstation clusters. Typical MPL programs look very similar to C++ programs with the keyword mentat used to distinguish classes which can be run in parallel. Mentat adopts a data-flow model of programming with the program graph constructed at run-time by observing data dependencies as the execution unfolds.

The Mentat Run-time System has an Instantiation Manager (IM) running on each workstation which handle scheduling and instantiation decisions of newly created Mentat objects. The IM uses the services of the Fully Automated Load Coordinator for Networks (FALCON), a heuristic scheduler based on a sender-initiated adaptive load sharing strategy. FALCON makes decisions on newly created work using a threshold policy. As per this policy, a task originating at a node is accepted for processing if the local state of the system is below a threshold. Otherwise an attempt is made to transfer the task to other suitable workstations. The workstation to which the task is migrated to is selected on a random, round-robin or best-most-recently basis. Once the work has been accepted for processing it cannot be migrated later to other processors.

1.2.2. Paralex

Paralex [1] is a parallel programming environment for distributed systems developed jointly at Cornell University and the University of Bologna. Paralex is also based on a data-flow model, with Paralex programs being composed of nodes and links. Programs are written using a graphical editor which specifies the dependencies in the program. It uses the ISIS toolkit to ensure uniform data representation and provides some fault tolerance and distributed debugging support.

Paralex maps computations by ascertaining dependencies in the task graph. Computations involving dependencies are grouped into *chains*. A chain is defined as a sequence of nodes in the task graph that need to be executed sequentially due to data dependence constraints. At load time, the run-time system schedules chains to workstations. Each workstation has a daemon process which constantly monitors the load. If two consecutive load measurements differ from each other by a certain threshold, the daemon broadcasts this information to an ISIS process group. Each Paralex process that wishes to, can collect load information by joining this process group and listening to load messages. Whenever the process controller for this ISIS process group becomes aware of the execution of a new node in the task graph, it examines future nodes in the graph that begin new chains. If they exist, the controller examines the load on the participating workstations and decides who should be in charge of the new chain. If a new mapping, different from the default mapping is proposed, then the new chain is started as per the new mapping and the previous default mapping is discarded. This is achieved by changing the coordinator for the Paralex chain. Thus, the run-time system constantly improves itself over the default mapping. The controller and the mapping process is never absolutely necessary for

program execution. If the controller fails, then new chains are simply spawned as per the default load time mapping. However, once a chain has been mapped to a processor it cannot be migrated until a new chain boundary is reached.

1.2.3. Dataparallel C

Dataparallel C is a SIMD style language [15] and run-time environment quite similar to C*. Dataparallel C was developed jointly at Oregon State University and the University of New Hampshire and currently runs on a variety of parallel machines and heterogeneous workstation clusters. The programming model is based upon virtual processors, global name space, and synchronous execution of a single instruction stream. Conceptually, the sequential portion of the Dataparallel C program executes on a front end, while the parallel portion executes on a large number of virtual processors. In its current implementation, each physical processor emulates the front end as well as its share of the virtual processors.

Load is balanced among heterogeneous workstations by varying the number of virtual processors emulated by each physical processor. At load time, the Dataparallel C compiler distributes virtual processors based on the relative speeds of individual workstations. Later work can be potentially migrated among processors whenever a new parallel portion of the Dataparallel C code is executed. To measure the rate at which each workstation is doing work, each workstation constantly monitors the average computation time per virtual processors emulated. Periodically, this information is shared with all other processors participating in the computations. The Dataparallel C run-time system estimates the time taken to perform an average load redistribution and then sets the time difference between two in-

formation exchanges so that the estimated load redistribution time is only a small fraction of the time between two information exchanges. After the load information exchange, each workstation decides how many virtual processors it should ideally be emulating. If this ideal figure is close to the actual number being emulated, no work migration is performed. In fact, if on any one of the processors, the difference is less than 5%, no work migration is initiated. Otherwise work is migrated by moving virtual processors so that the number emulated on each physical processor is equal to the ideal, calculated number.

1.2.4. Dome

Dome is a parallel programming environment [2], developed at the School of Computer Science of Carnegie Mellon University by Adam Beguelin and others. Dome was built for heterogeneous workstation clusters and is actually an acronym for Distributed Object Migration Environment. The run-time system and language is built on top of PVM [8]. The Dome language is object based and data parallel in design. It uses a run-time environment which provides support for architecture independent checkpoint and restart.

Dome balances load by performing periodic synchronizations and data migration among parallel tasks. Load balancing phases in Dome are triggered when a certain number of operations are over on Dome objects. During a load balancing phase, tasks exchange information regarding their performance during the previous work phase. Dome tasks communicate with each other using a ring topology. Each task exchanges its performance information with its left and right neighbor in the ring. Based on this information, portions of Dome tasks are migrated. The data

movement is therefore only between neighbors and so expensive global synchronizations can be avoided. However, since only local synchronizatons are performed, the rate of convergence to the ideal distribution is slow.

1.3. OVERVIEW

Load distribution strategies can be classified into different categories depending on their nature and method of work migration. Chapter 2 examines some of these differences and classifies the different strategies under some broad categories. A complete and effective load distribution strategy should have some components which ensure that it gives good overall performance under all conditions of load. These components are discussed in detail in this chapter. The Charm Parallel Programming Environment and Run-time Environment are discussed in Chapter 3. Chapter 4 examines the actual strategy used by the load distribution scheme to schedule and migrate work to ensure that all workstations finish off available work as close as possible to each other. The load distribution strategy was tested on a series of application test programs. The programs in this test suite are described in Chapter 5. Chapter 6 describes the evaluation methods used to prove the effectiveness of the load distribution strategy and various monitoring systems to gather data and convert it into intelligent information through graphical animations. Chapter 7 summarizes the results obtained and demonstrates the effectiveness of the load distribution strategy. The report ends with a brief summary of the work done which includes its advantages, shortfalls and possible avenues for future development.

2. LOAD DISTRIBUTION SCHEMES

2.1. MOTIVATION

Workstations of different processing speeds and architectures have different performance characteristics. In addition, multiple jobs and users make workloads on heterogeneous workstations highly variable. It is therefore very difficult to predict the total time taken by a parallel job if no dynamic scheduling or load sharing is performed. Several studies have shown that substantial performance increases can be obtained [5] [13] when intelligent dynamic scheduling is used.

2.2. SCHEDULING STRATEGIES

Tasks can be scheduled in two ways. *Static Scheduling* refers to scheduling decisions made at compile-time using *a priori* knowledge of the system while *Run-time Scheduling* uses current system state information to make scheduling decisions. In general, run-time scheduling gives much better performance increases [17]. Run-time scheduling algorithms have the potential to outperform static algorithms because of the use of system state information which improves the quality of their scheduling decisions.

2.2.1. Static Scheduling

Static scheduling divides the work based on decisions at compile time and does not adapt to changing network conditions. It does not give very good performance increases due to a number of reasons.

- Task Sizes: Task times can vary dramatically from predicted compile-time estimates due to system load and un-predicted program behavior.

- Number of Processors: Allocation of tasks at compile time is based on assumptions that a certain number of processors will be available for use which need not always be the case.

- Branch Conditions: Due to the unpredictability of branch conditions at compile time, actual times can vary from forecasted estimates.

- Depths of Recursion: In some problems, especially divide and conquer problems, unknown depths can increase task sizes as well as control the dynamic creation of other tasks.

- Overheads: Network delays and memory latencies are very unpredictable and may cause less or more overhead than anticipated.

2.2.2. Run-time Scheduling

Run-time scheduling strategies, based on scheduling of tasks at load-time usually perform much better because of their adaptive nature. These strategies use current system state information to influence their scheduling decisions.

- Load Sharing: Dynamic scheduling schemes usually employ a load sharing strategy that shares the workload among participating processors.

- Utilization: Since dynamic scheduling schemes are better equipped to distribute the available work, overall utilization of the workstation cluster will be higher.

- Adaptability: Dynamic scheduling strategies are usually designed to adapt to varying loads on workstations.

2.2.3. Workload Descriptors

A suitable index of workload on a workstation is a key issue in determining the efficacy of scheduling and dynamic load balancing heuristics. The *load index* of a workstation is a measure which indicates the performance of a task when executed on it. A number of such measures of load have been proposed [13]. To be effective, load index readings taken when tasks initiate should correlate well with task response times. Commonly used indices are length of task queue length, available free memory, context switch rate, system call rate, CPU utilization, idle time, and length of run queue over the last one minute. Combinations of these descriptors can also be used as indices of load. However studies [13] have indicated that the combinations may fare *worse* than simple descriptors.

Various run-time systems use some of these descriptors. The V-system [17] uses CPU utilization. It measures this by running a background process, with the lowest priority, that periodically increments a counter. The counter is then periodically polled to find what proportion of the CPU has been idle. Condor [17] uses idle time to detect potential workstations. It transfers work only if the workstation has been idle for 12.5 seconds. A local scheduler then checks for user activity every 30 seconds. On detection of user activity for two consecutive 30 second intervals, the scheduler preempts the task after saving its current state. If the owner remains active for 5 minutes or more, the foreign task is transferred back to the originating workstation. The Stealth [17] Distributed Scheduler relies on CPU utilization and available memory to migrate tasks.

2.3. DYNAMIC LOAD DISTRIBUTION

Despite intelligent scheduling methods, due to the variable nature of workload on workstations, some form of task migrating mechanism must exist to properly utilize the capabilities of workstations by transferring work from heavily loaded workstation to more lightly loaded ones. Two general methods of deciding when to transfer work exist. *Load Sharing* is a method by which load is transferred from heavily loaded workstations to idling workstations. Usually, *anticipatory transfers* are made from heavily loaded workstations to workstations which have a high probability of idling soon. Thus, system utilization is enhanced by ensuring that the participating workstations idle for the minimum possible time. *Load Balancing* algorithms [5] go further by trying to equalize the load on all workstations. Load balancing can potentially reduce the standard and mean deviation of task response times, relative to load sharing algorithms, but the higher transfer rates can potentially outweigh the performance improvements.

Typically, a load distribution algorithm has several components. These components together ensure the proper functioning of the load distribution strategy. Usually, load distribution strategies are centralized or distributed or a combination of both. Centralized policies use one workstation, called a controller or scheduler, which decides when and how to transfer work. In distributed policies, workstations individually and independently handle their task migration decisions. However, they might rely on information received from other workstations to improve the quality of their decisions. Work transfers can be preemptive or non-preemptive. Preemptive transfers involve transferring partially executed tasks. This operation is quite ex-

pensive, involving storing the current virtual memory image, process control block, un-read I/O buffers and messages, file pointers, timers that have been set and so on [17]. Non-preemptive task transfers however, involve tasks that have not yet begun execution and hence do not require transferring the task's current state. Usually, load distribution strategies have the following components :

- Information Policy
- Location Policy
- Selection Policy
- Transfer Policy

2.3.1. Information Policy

Information about participating workstations must be disseminated to others to help workstations make intelligent decisions on whom to ask for work. The information sent must indicate in some way the current processing speed of the workstation and the amount of work left in the task queue of the workstation. The information policy decides when and whom to send this information.

Demand driven policies make a workstation collect information from other workstations only when it either becomes a sender or a receiver, thereby making it a suitable candidate to initiate load transfers. Demand driven policies can be sender, receiver or symmetrically initiated. In *sender-initiated* policies, senders look for receivers to transfer their workload to. *Receiver initiated* policies, on the other hand, solicit offers from potential senders. *Symmetrically initiated* policies use a combination of both, with load sharing decisions being initiated by both heavily loaded senders or idling receivers.

Periodic policies, either centralized or decentralized, send information at predefined intervals. Periodic policies are not very adaptive. The benefits of sending status information when all workstations are heavily loaded is minimal because all processors have enough work and are busy finishing off pending work. In fact, sending status information can just slow down the over-all processing speed of the run-time system due to the increased individual workloads caused by sending and receiving messages at individual workstations.

State driven policies operate by sending status messages only when the workload status of the workstation changes by a certain predetermined amount. Under a centralized scheme, the workstations would report their current state to a centralized collecting point while a distributed policy would send the new state information to its peers.

2.3.2. Location Policy

The location policy decides which workstation to query for work or transfer work to. Centralized policies react by querying a controller about workstations which want to get rid of extra work or want some work to prevent idling. Decentralized policies use methods like *polling* where a workstation sends requests to all its peers for work. Polling algorithms sometimes use a system of bids to select potential workstations. This system works quite well under conditions of low and medium load. It however performs poorly when the system is heavily loaded, the reason being that at higher system loads, the overhead involved in finding a destination processor may outweigh any performance gains. *Drafting* algorithms on the other hand select one heavily loaded processor and drafts it by requesting work from it. This scheme performs well under heavy loads but suffers from over-draining of

resources from the heavily loaded processor due to being drafted by several idling workstations. Thus there is a potential for over-migration of jobs.

2.3.3. Selection Policy

Selection policies decide what tasks should be migrated and what should not be. It often makes sense to migrate computationally intensive tasks while tasks that are not computationally intensive are usually better off executed on the same processor that created it. Unfortunately, it is not easy to predict what tasks will be computationally intensive and what tasks won't be. However, intelligent compilers or the programmer's explicit directives can be used to generate some information about the type of the task at its creation. This information is later evaluated by the selection policy to select tasks. Most load balancing schemes use much simpler methods. Almost all of them only migrate tasks that haven't begun execution to avoid expensive techniques like checkpointing or transferring the process image information. One of the most commonly used strategies is to transfer tasks in FIFO order. Thus, tasks that have been waiting for the maximum time in the task queue get transferred first. A variant of this scheme is to use the LIFO order where the tasks that came in last are migrated first. Such a scheme can be useful for heuristic searches where the latest tasks probably have the best chance of delivering results. Another important criteria for the selection policy is the amount of data that needs to be transferred for the task to run to completion on the destination processor. If the amount of data to be transferred is large, the time spent in moving the data may offset any performance gains.

2.3.4. Transfer Policy

The transfer policy determines the nature and method of work transfer. Some transfer policies are receiver initiated while others are sender initiated. Receiver initiated policies are activated when a workstation is idling for lack of work or is about to idle for lack of work. Sender initiated policies on the other hand are triggered when the load or number of tasks on a machine exceeds certain limits. Such policies do not perform well in conditions of high overall load [4] but are quite effective in low and medium load cases. Receiver initiated policies, however, are more effective in cases of high overall load compared to low and medium load conditions. The method of transfer often varies from algorithm to algorithm. Simple schemes grab work or offload work without permission from the other workstation. More complicated schemes are more cooperative, doing the transfer based on some mutual agreement reached beforehand. This might however involve a handshaking format and the additional overhead involved may not be worth the performance improvements. *Symmetrically* initiated transfers are used by load sharing schemes where work transfers are initiated when a node becomes a sender as well as a receiver.

3. THE CHARM PARALLEL PROGRAMMING ENVIRONMENT

3.1. LANGUAGE SPECIFICATIONS

Charm is a machine independent, parallel programming language and run-time support system first developed at the University of Illinois at Urbana-Champaign. The language is based on an object based, message driven, MIMD paradigm [6]. The most basic object capable of parallel work is called a *chare* (an archaic synonym for chore). Chares can be spawned and executed by the programmer. A chare definition consists of local variables and a set of entry point functions. It may also include private functions. Each chare has a unique *id* on execution. Chares can also send messages to and from themselves on knowing the relevant chare id. Messages to chares are received at *entry points*, which are specified portions of a chare that are capable of receiving messages. On receiving a message at an entry point, the code associated with the entry point is executed. After execution, chares switch to a suspended stage. Suspended chares can be made to wake up by sending messages to specified entry points within that chare from other live chares.

From a programmer's point of view, the system operates with a pool of messages. These include messages specifying the creation of new chares as well as messages for existing chares. Each processor picks up a message from this pool, executes the entry point indicated by it, possibly modifying the local variables of the associated chare instance and depositing new messages in the systems pool. It then returns to pick another message from the pool. Two messages directed to the same chare are not concurrently executed. Also, execution at an entry point on a

processor is not interrupted to execute another entry point on the same. Messages to chares are sent using the `SendMsg()` as well as other calls all of which are non-blocking. It is also worth emphasizing that although Charm provides a number of *send* calls, it provides no corresponding *receive* calls. Instead, execution is driven by the existence of arrived messages.

Various information sharing abstractions and high level primitives exist in Charm. *Read Only* variables are variables whose values are initialized at the beginning of the computation. The chares on any processor can access this value. The system may implement this as a single shared copy on shared memory machines or as a private copy on individual processors. Read only variables are the only global variables permitted in Charm. *Distributed tables* is a collection of [key,data] pairs. It supports *insert*, *find* and *delete* operations. As the name suggests, the pairs can be distributed across processors and a call to find data associated with a particular key, results in data being send to a specified entry point within a named chare. *Accumulators* are shared data structures which are useful to maintain global totals. Finally, *branch-office chares* are high level primitives that are basically chares which are replicated or has branches on all processors. Branch-office chares are typically used for distributed data structures and for handling local services.

Charm programs are written as modules so that they can be reused or called from other modules with minor modifications. Typically, every Charm program has a *main chare* from which all program execution begins. Subsequent tasks are created dynamically from this chare. These tasks can be chares or branch-office chares. To illustrate this process, the following Hello World program is written using branch-office chares. On execution, the program will printout the string *Hello World* on

all participating workstations. The entry points `DataInit` and `ChareInit` on the main chare are executed first, in order. On receiving the initialization messages, the entry point `Print` on the branch-office chares are executed on each processor. After execution, the termination detection algorithm detects the finish of all useful activity and therefore returns control to the programmer by calling the `QUIESCENCE` entry point at which the programmer decides to terminate execution.

```
module HELLO {
chare main
{
 int BocNum;

 entry DataInit:
  {
     DummyMsg *msg;
     msg=(DummyMsg *)CkAllocMsg(DummyMsg);
     BocNum = CreateBoc(hello, hello@Print, msg);
  }
 entry ChareInit:
  {}
 entry QUIESCENCE:
  {
    CkExit();
  }
}
```

```
BranchOffice hello
{
 entry Print:(DummyMsg *msg)
  {
    CkPrintf(" Pe[%d]: Hello World \n",McMyPeNum());
  }
}
} /* module */
```

Charm can also be used as a as a universal back-end for visual programming tools like Dagger and DP-Charm - a data parallel language developed on Charm which implements a subset of the official HPF (High Performance Fortran) language. Charm programs are translated into C by a Charm translator. Entry points within a chare usually execute C code. Charm programs are essentially a superset of C excluding static and global variables. The chare instances themselves are simply data areas within a Charm process and most high level Charm abstracts get translated into C structures.

3.2. RUNTIME ENVIRONMENT

The Charm environment runs on several shared memory and distributed memory machines including the NCUBE, iPSC/860, Sequent, Meiko CS-2 and the CM-5. The environment is also available for a network of SPARCStations. The Charm environment was subsequently ported to run on HP and IBM RS6000 workstations at Oregon State University. Charm, when running on workstation clusters,

uses TCP/IP and UDP for all inter-workstation communications with the majority of it being UDP. Since UDP does not provide guarantees on inter-processor communications, a sliding window protocol is used to ensure that communication in Charm is guaranteed. The run-time system also automatically handles its own memory management and dynamic load balancing.

The Charm run-time system for workstation clusters was modified at Oregon State University to better support truly heterogeneous computing. During startup, the Charm run-time system looks up a file to find information on how to run the Charm program in parallel. This information includes the names of the workstations, login names, relative speeds of the machines, password, path-name directions on where the Charm run-time system can be located on that machine and model number of the workstation. Since the Charm run-time environment makes no assumptions, it is possible to run programs on machines where the user does not have the same password or login names and not supported by a Network File System. In fact theoretically, with the available information, we can run a Charm program on workstations that are located anywhere on the Internet. Even if the machines are on the same local area network and with the same Network File System, the Charm system allows simultaneous compilations of the source code for each of the different machine architectures. The run-time system is also intelligent enough to locate the correct binary (depending on the machine architecture) to execute. The Charm run-time system takes care of various housekeeping facilities like termination detection and dynamic load balancing by in-built branch-office chares.

The dynamic load balancing module [18] on the distributed memory version of the Charm is implemented by a Load Balancing branch-office chare (LDB-BOC).

A load balancing BOC has one instance resident on each processor. It provides entry points and function calls and interacts with other Charm system BOCs to dynamically balance the load. Each processor creates and initializes the LDB-BOC. At the BranchInit entry point, the number and names of neighbors are recorded and stored in the data area of the BOC. This depends on the interconnection network amongst the processors. However, in a workstation cluster, since all workstations can communicate with all other workstations, we can personalize the neighbor list according to the machines we would like to use and the neighbor connections desired. This neighbor information is later used by the LDB-BOC to ask other processors for work or to exchange status updation messages with. The load balancing strategy decides which tasks should be relocated on other processors and when this needs to be performed. The load balancing BOC is designed in such a way that it can call different load balancing strategies by linking the Charm library with the correct load balancing object files. So we can very easily use the Charm run-time system as a test-bed for different load balancing strategies. A number of load balancing strategies are provided with the Charm distribution. These include placement of chares randomly, on a priority basis and on other more complicated strategies.

4. THE LOAD DISTRIBUTION STRATEGY

4.1. LOAD SHARING ON THE ETHERNET

When load sharing on the Ethernet, the *sine qua non* is to balance work and exchange load information only if it is absolutely necessary to do so. The high message latency and low network bandwidth force load sharing strategies to be effective only if they transfer or exchange the minimum amount of information. An efficient run-time scheduling portion can help by ensuring that only a minimal amount of work needs to be transferred by the dynamic load sharing portion of the load distribution scheme. The initial distribution is done in a centralized manner from one node, referred to as the *scheduler*. Subsequent dynamic load sharing is performed in a distributed adaptive fashion, based on status information exchanged between neighboring nodes. Dynamic requests for more work are triggered only when a processor is about to idle for lack of work. No periodic global distribution of work is performed as this potentially wastes network bandwidth, delays computation, encourages constant migration of work and promotes back and forth exchanging of work popularly known as the *ping pong* effect. Another important feature of the load sharing algorithm is that all status information exchanged between neighbor nodes is in terms of time units. In addition, work requested by a node, when it runs out of work, is calculated on the basis of time units. This potentially eliminates inconsistencies or errors due to the variable nature of work in MIMD computation. Each unit of work is referred to as a *chare* in the Charm environment used to implement the load sharing scheme.

4.2. RUN-TIME SCHEDULING

Heterogeneous workstations have varied processing speeds. The rate at which they do work depends on factors like the clock speed, CPU, math co-processor and compiler optimizations. Generally, SPEC_INT92 and SPEC_FP92 ratings are used to categorize the performance of a workstation for integer and floating point calculations respectively. Parallel application programs typically use both integer and floating point operations. This means that rating machines solely on SPEC_FP92 or SPEC_INT92 will not accurately reflect their processing speed for a specific application program. However, it is difficult to measure the actual percentages of integer and floating point operations in a typical parallel application. Since parallel applications are run a large number of times, it is often a worthwhile investment to run each application with a reduced problem size *once* for each type of workstation that will take part in the computations. This is will represent the best measure of the actual rate of processing of the workstation for the application's unique mix of integer and floating point operations. The ratios of the time obtained for each workstation can be used as the relative processing speed of the workstation for that application. When measuring the processing speed of workstations, care must be taken to see that the workstation is running no other computation or I/O intensive tasks. Distributing work based on processing speeds of workstations would have been enough if they were available for dedicated use. However, most workstations operate in a multi-user environment running both interactive as well as computation intensive jobs. Hence, the *actual* available processing power of a workstation is dependent on the processing speed of the workstation under conditions of no or very light load as well as the current load. The emphasis of the current load on scheduling should not be underestimated. The presence of just one another computation intensive job

will effectively *double* the total time taken since now only half of the CPU cycles are available for use.

Various effective and useful measures of load have been proposed. The most commonly used indices are the average task queue lengths, available memory, context switch rate, system call rate, CPU utilization or a combination of these. Most of the above information is stored in kernel memory and the user needs special privileges to access them. The time taken to access this information involves expensive system calls and varies dramatically, based on the operating system, vendor and hardware CPU used. Hence, in the interests of portability and un-obtrusiveness, simple methods of measuring load are often used. Studies [13] completed in the past have also indicated that using complex measures of load have little or no performance gains over simpler load indices. Moreover, the best indicator of load was found to be the task queue length.

Definition 1 *The load on a workstation is defined as the average number of runnable processes over the past one minute incremented by one.*

Runnable processes are considered to be those processes that can possibly run at that instance if enough CPU resources were available and excludes processes which have been stopped or are waiting for I/O. This measure was used due to its easy availability through standard UNIX commands like `uptime`. Load is incremented by one to take into account the additional load caused by the new Charm process.

4.2.1. Processing Capacities

The run-time scheduling phase of Charm is based on the *Relative Processing Capacities* of participating workstations. The run-time scheduler distributes tasks initially to all workstations based on this information. This information is obtained from archived data about machines and from polling workstations about their status at the onset of computation.

Definition 2 *The Relative Processing Capacity (RPC) of a workstation is defined as the ratio of the relative processing speed of the workstation to the load on that workstation.*

$$RPC = \frac{Relative\ Processing\ Speed}{Load}$$

Definition 3 *The Relative Processing Speed (RPS) of workstation i and application program p, represented by RPS(i,p), is calculated by measuring the time T(f,p) needed by the fastest workstation and dividing it by the the time T(i,p) needed by workstation i to complete the same application program with a reduced problem size.*

$$RPS(i,p) = \frac{T(f,p)}{T(i,p)}$$

Since this number varies for each workstation and application program, the values have to be calibrated once for each workstation and application if high efficiency and throughput are needed. However, if high efficiency is not desired, heuristic values based on the estimated mix of integer and floating operations can be used along with SPEC_INT92 and SPEC_FP92 ratings. In such cases the Approximate Relative Speed is defined as follows in terms of Processing Speed (PS) for the workstation i, the fastest workstation f and application program p.

Definition 4 *The Approximate Relative Processing Speed (ARPS) of workstation i and application program p is defined as the ratio of PS(i,p) to PS(f,p).*

$$ARPS(i,p) = \frac{PS(i,p)}{PS(f,p)}$$

where $PS = \%\ of\ integer\ ops \times SPEC\_INT92 + \%\ of\ FP\ ops \times SPEC\_FP92$.

The Charm run-time environment on startup determines the Relative Processing Capacities of all participating workstations. It then lets the scheduling processor, know these rates. When work or chares are spawned on the scheduling processor the run-time system automatically takes care of chare distribution based on the RPCs of participating workstations. In Charm, where work is almost always spawned dynamically, it is difficult to predict at compile-time exactly how many chares will be spawned. The scheduling strategy therefore assumes that a minimum quantum of chares will be spawned and schedules the spawned chares based on this quantum. If more than the quantum of chares are actually spawned, the scheduling strategy schedules additional chares assuming another quantum of chares will be spawned. The Charm environment schedules work spawned only on the scheduler. Work spawned on other processors is always enqueued locally.

4.2.2. Scheduling Hints

Ideally, the scheduling strategy should be totally transparent to the application programmer. The above implementation makes this scheduling user transparent. However, sometimes programmers may want to achieve better performance from the scheduling strategy. The Charm scheduling strategy adapts by listening to scheduling directives from the user and trying to act accordingly. These hints are not required to be followed. The user is therefore cautioned never to program

assuming that his directives will always be followed implicitly. The distribution of chares can also be important to achieve good performance. If for example, neighboring chares communicate more often, the programmer can profit by distributing adjacent chares on the same processor based on their RPCs. Currently, directives exist for distributing chares contiguously, interleaved or not to schedule at all (enqueue locally). Contiguous distribution implies adjacent chares are scheduled to the same processor while interleaved scheduling distributes chares in a round-robin fashion. Distribution of chares in strides (say for example, every sixteenth chare scheduled to the same processor) and other similar strategies can also be achieved through clever use of these directives. Additional directives also exist to change the predefined quantum of chares that the scheduling strategy expects. This will ensure a much better distribution than by the default distributions, since the scheduler now knows precisely how many chares will be spawned and can schedule accordingly. The following snippet of Charm code shows how such directives can be used to effectively schedule tasks using the `LdbHint` calls.

```
module MultiDistribution {
chare main {
 entry DataInit:
  {}
 entry ChareInit:
  {
   int i;
   DummyMsg *SndMsg;

   LdbHint(INTERLEAVED,512);
   for(i=0;i<512;++)
```

```
    {
     SndMsg=(DummyMsg *)CkAllocMsg(DummyMsg);
     CreateChare(Compute, PhaseI@Compute,SndMsg);
    }
    LdbHint(CONTIGUOUS,512);
    for(i=0;i<512;++)
    {
     SndMsg=(DummyMsg *)CkAllocMsg(DummyMsg);
     CreateChare(Compute, PhaseII@Compute,SndMsg);
    }
   }
}

chare Compute {
 entry PhaseI:(DummyMsg *msg)
  {...}
 entry PhaseII:(DummyMsg *msg)
  {...}
}
}
```

4.3. DYNAMIC LOAD SHARING

The dynamic load sharing component of the load distribution strategy is a distributive, adaptive, receiver driven strategy customized for efficient performance on the Ethernet. Since the dynamic component is co-operative and non-preemptive, work is transferred only on mutual agreement. This innovative method ensures

transfer of work is performed only when needed and reduces chances of unnecessary migrations of work. The strategy has two main components.

1. The Status Exchange Component
2. The Work Transfer Component

4.3.1. Status Exchange

The key to efficient dynamic load sharing is to ask for work intelligently. Work must be requested from processors which are more likely to have excess work. It therefore becomes imperative that the dynamic load sharing scheme have a status exchange component which periodically informs its neighbors regarding its current status. The status information exchanged by each workstation is its *Forecasted Finish Time.*

Definition 5 *The Forecasted Finish Time (FFT) of a workstation is the time estimated by it to finish off the remaining tasks at its current rate of execution.*

The value of FFT will vary, depending on the amount of work left and the current rate of execution. The amount of work left in the queue is periodically monitored and the execution rate re-calibrated at predefined intervals of time. This implies that the FFT constantly changes, reflecting any variations in the current workload. The execution rate is measured in milliseconds per chare. The total time elapsed and not user time is used for calculations and so the rate measured also takes into account the effect of other processes running on the workstation. By using the total time elapsed, the actual rate at which the node is processing work can be measured and not just the rate at which it is executing the program.

In an Ethernet scenario where every workstation can communicate with every other workstation it is easy to flood the network with status information. In a cluster having n workstations a total of $n \times (n\text{-}1)$ messages can be exchanged for *each* status update. As the value of n increases, this progressively becomes a very unattractive way of conveying status information. To prevent this, the user is given the flexibility to specify a neighbor file which contains a valid arbitrary network interconnection, instead of the default fully interconnected model. This prevents an inordinate number of status messages from flooding the network. Dense graphs can also be used to reduce overhead by minimizing the number of status messages. The neighbor connections determine which machines to exchange status information with and ask/receive work from. Typically, neighbor nodes exchange status messages and request work only among themselves in case one node idles before some of its neighbors. Status information is also *piggy-backed* on every normal message sent between processors. This ensures that status information is given a free ride with conventional user messages.

The algorithm developed further ensures that that not too many status messages are sent by eliminating frequent updates and non-useful status messages. Typically, status messages are needed only when a node has run out of its initially assigned work. This usually happens at the very end of computation. The reason for this is that if the initial scheduling phase does its job well, nodes will not idle for lack of work until all nodes are very near the end of their initially allocated work. Thus, the probability of a neighboring node idling is inversely proportional to its FFT. This implies that few messages are needed when nodes have lots of available work and more frequent messages when nodes have less work or are close to finishing off their initially assigned work. Therefore, each workstation will have a more

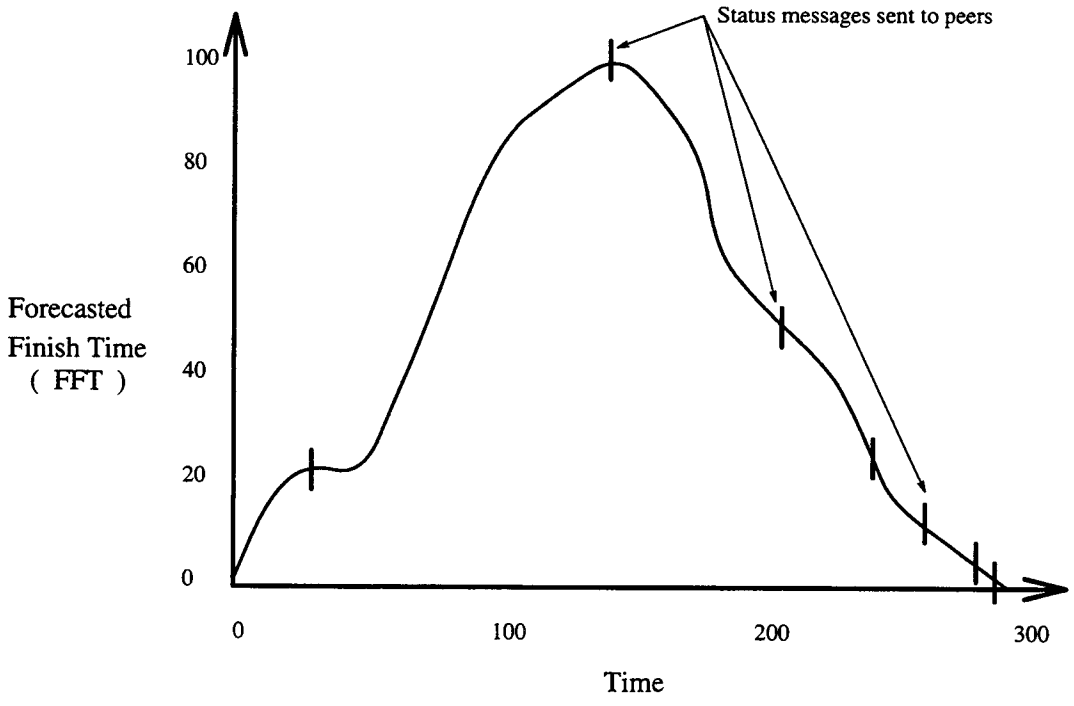

Figure 4.1. Forecasted finish time graph

accurate representation of its neighbors' Forecasted Finish Time when it probably needs it the most. This is accomplished by sending no status messages when nodes are receiving their scheduled portion of work by keeping track of the work queue. While this queue length is increasing, all status updation messages are suspended. After receiving all its work, every node calculates the rate at which it is processing chares as described previously. Based on this rate it predicts its FFT. When the queue length starts to show a decreasing trend, the current FFT is sent to all its neighbors. Whenever this Forecasted Finish Time is halved due to processing of chares, the next group of status messages are sent to its neighbors (Fig 4.1) where it is stored in a database. This method of status message updations results in fewer messages in the beginning and more frequent messages at the end of computation.

4.3.2. Work Transfer

When the Forecasted Finish Time of a node falls below a certain threshold value, the node sends out a request for more work from one of its neighbors. This threshold value will greatly depend on the type of underlying network used. The decision to choose a particular node is based on a decision algorithm that decides which of its neighbors to ask for work. This receiver initiated strategy is superior to other sender initiated strategies because the receiver can best determine when it is about to idle. The main goal here is to reduce idle time on each workstation to a minimum. Any scheme which reduces idle time on a workstation to the minimum possible will indirectly yield a balanced workload.

Location Strategy

When the FFT of a workstation falls below a certain threshold, the location strategy is invoked. The location strategy looks up an internal database it maintains for all its neighbors to determine whom to ask for work. The neighbor with the maximum FFT is selected as the target. Confirmation checks are then performed to determine whether this value is greater than a prescribed minimum. If the maximum FFT of the neighbor is less than this minimum, no requests for work are sent. This is to ensure that the inconsistencies between status information and actual work left does not make a workstation search for work from unworthy workstations. When the maximum FFT of all its neighbors is lower than the prescribed minimum, the decision strategy forces the Charm environment to enter a shutdown mode called *Quiescence*. However, if new work gets created or arrives from a neighbhor, the location and status exchange components reawaken and start functioning again.

Work Migration Strategy

Once a node has decided on a neighbor to ask for work, it tries to estimate how much work it should ask for, so that it can do useful work for the next predefined interval of time without asking for more work in between. This is done by finding out the number of chares it can process at its current rate of execution in this time interval. Ideally, this quantum of time is the maximum time difference tolerable between the execution times of two neighbor nodes. This quantum of time should neither be too long or too short. If this quantum is too large then too much work will be migrated and will also encourage the ping-pong effect. If this quantum is too small, it will encourage a large number of frequent requests. For the medium grainsize problems that Charm is developed for, a quantum of size 5000 milliseconds was found quite suitable for an Ethernet environment. If the computation time of each chare, based on its current processing rate, is greater than this time, only one chare is requested for. Thus, whenever a workstation is about to idle and on satisfying other previously mentioned conditions, a request for at least one chare will always be sent.

Once the number of chares is determined, a request is sent to the correct workstation. The workstation on receiving the request message, determines how many chares *it* can process in the same predefined interval of time at *its* current processing rate. If it can process all its available work in the same period of time, it will reject the request for work. Otherwise, it reserves those chares for itself. Of the remaining chares available, it finds out the number of free chares. A chare is considered to be a free chare if the user does not specify that that chare has to

be executed on a specific machine. Also, once a chare has begun execution and is temporarily suspended it loses its free chare status. If it can spare the number of free chares requested for, the node then proceeds to transfer these chares to the workstation that originated the request. However, if fewer chares than asked for can only be spared, only those are sent. Thus the amount of work received by a workstation will always be less than or equal to the amount of work asked for. This ensures that work is transferred only with the joint cooperation of both the sender and receiver. The updated FFT is also piggy-backed on the chares transferred so that the request workstation can be quickly made aware of its new status. Each chare keeps a count of the number of machines it has migrated through. This count is referred to as the number of *hops* the chare has so far taken. If a chare has hopped too much it is *grounded.* Once a chare is grounded it is removed from the free chare list forcing it to be executed at the current machine. This history of hops prevents work from infinitely moving back and forth between processors.

Retry Strategy

If a node is refused work from a workstation, it updates its database and its record of the FFT of that workstation as zero. This is done to prevent the workstation from asking the same workstation for more work until there is a considerable change in the transmitted FFT of that machine. The decision algorithm is again invoked to detect the next suitable processor to ask for work. If a suitable candidate is found, the work transfer strategy is initiated again. A workstation will retry for work only a fixed number of times, after which it assumes all nodes are idle and the job completed. The environment then proceeds with the initiation of the Quiescence state. However, if a workstation receives work in one of the retry attempts, it

erases all its history of retry attempts and starts afresh. This means that successive successful attempts are not considered retry attempts. If however, a workstation cannot detect a suitable candidate workstation that satisfies all the criterions of the decision strategy during a retry attempt, it halts the retry mechanism and goes into the shutdown mode.

5. APPLICATION PROGRAMS

5.1. TEST SUITE

A suite of application programs was developed to test the efficiency of the load distribution strategies. All the application programs were programmed in the Charm programming language. These programs were run using several variations of the load distribution strategy to evaluate their relative merits and deficiencies. This set of programs are fairly computationally intensive and mainly involve graphical and numeric results.

- Matrix Multiply
- Raytracing
- All Pairs Shortest Path
- 2-D Morphing

5.1.1. Matrix Multiply

This matrix multiplication algorithm uses two variable sized matrices which are multiplied in parallel and the results sent to one processor which stores it in the result matrix. A general matrix multiplication is of the form $C = A \times B$ where C represents the resultant matrix. In this particular algorithm, the entire B matrix is transposed and replicated on all participating processors. Matrix B is stored in its transposed form to improve cache hits during the actual multiplication. Each row of matrix A is split into several parts and distributed to processors. The size

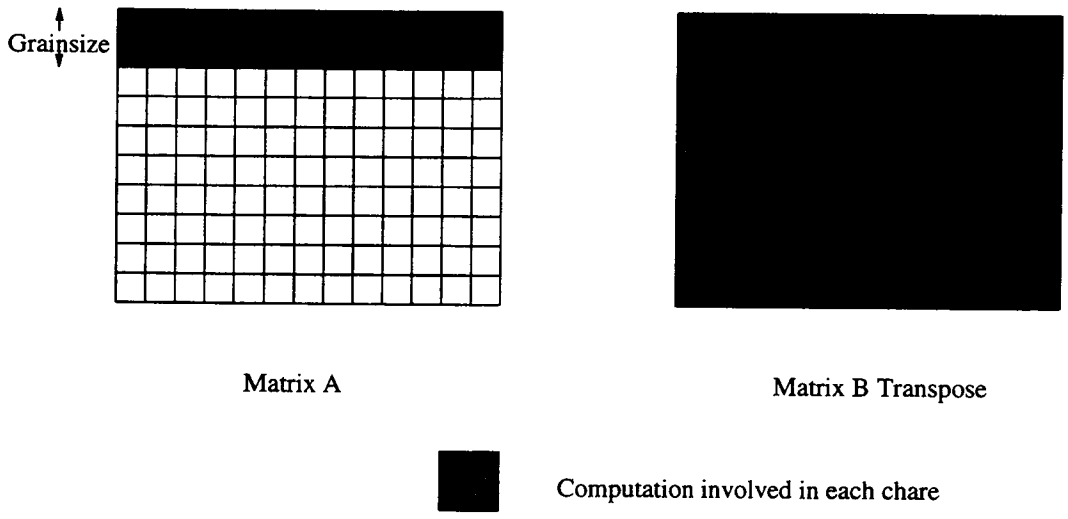

Figure 5.1. Matrix multiplication

of each part determines the grain-size of each task (Fig 5.1) . If each row is split into p subparts, the number of multiplications per task will be $(n/p)^2$, where n is the number of columns of A. If each row of A is split up into parts, only a partial result will be obtained and all these partial results have to be summed up finally to obtain the final C matrix. Since B is replicated and the relevant data of A is attached to each task, the task can be migrated to *any* participating processor by the load sharing strategy and the actual calculation performed on it.

5.1.2. Raytracing

Raytracing is a popular method for generating very realistic 3-D images (Fig 5.2). Unfortunately this process involves using a large number of computationally intensive operations. Ray tracing proceeds by determining the visibility of surfaces by tracing imaginary rays of light from the viewer's eye to the objects in the screen [7]. A center of projection and a window on an arbitrary view plane are selected.

This window can be considered as a rectangular grid, each of whose elements correspond to a pixel at a desired resolution. These pixels are set to the average of the current intensity values of all the rays as they pass through the pixel after reflections and refractions on objects in the scene to be raytraced. The rays from the eye are traced as they are reflected and transmitted by objects. The reflected and transmitted rays continue to be traced until a maximum tracing depth is reached or the rays no longer hit any object. Since each point on the screen can be computed in parallel with other points, raytracing is ideal to be parallelized.

The primary basis for raytracing is the fact that when a ray hits our eyes we see its intensity, which simply stated is its color [16]. This color is what is left of the original intensity of the ray after it came from the light source. Whenever the light ray hits an object, parts of its wavelength are partly or totally consumed. Only the remaining wavelengths manage to escape, scattered in different directions. One of these escapees of this second scattering, manages to reach the viewers eyes and produce the sensation of sight. Thus each ray has a history that formed its intensity. However, given a light source, it is difficult to determine whether a specific ray will finally reach the viewer's eye. So the reverse of the actual process is performed to minimize unnecessary computations. Rays are therefore pretended to originate from the viewer's eye and the entire raytrace is done in reverse. Simple schemes consider only reflected light while for better results the refractiveness and transparency of objects have to be considered.

Several public domain packages exist which provide excellent raytraced results. POV Ray (Persistence Of Vision Raytracer), Craig Kolb's Rayshade and RTrace are some examples. Rayshade, which is from Princeton, was used for the

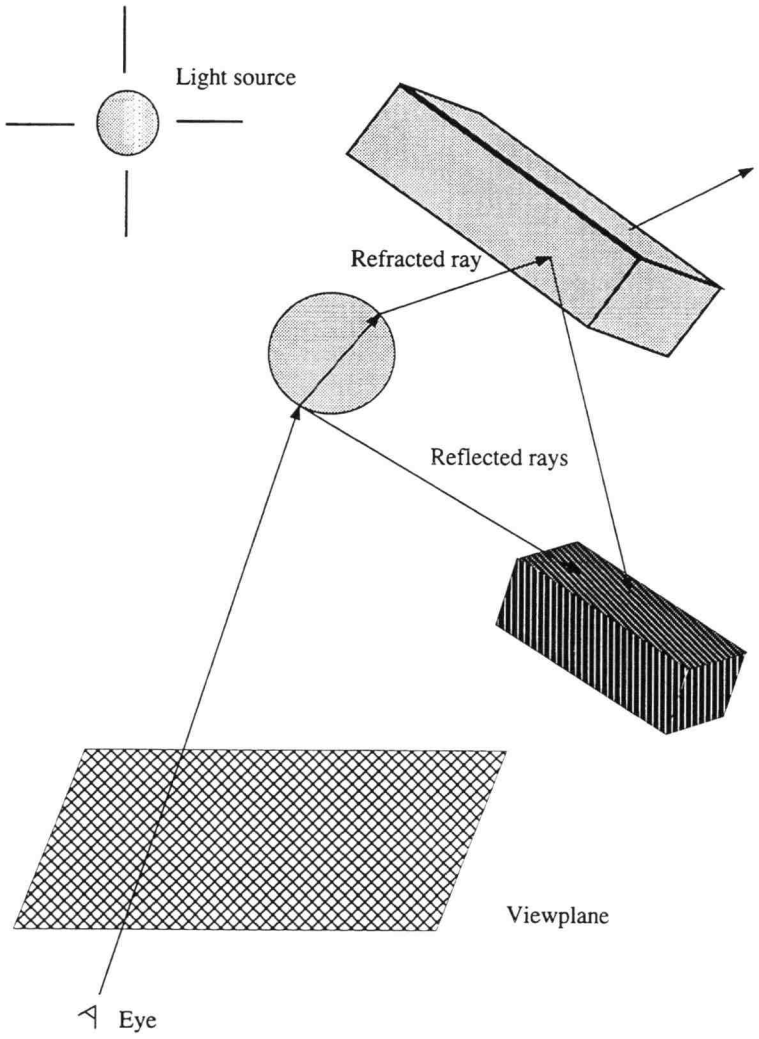

Figure 5.2. The process of raytracing

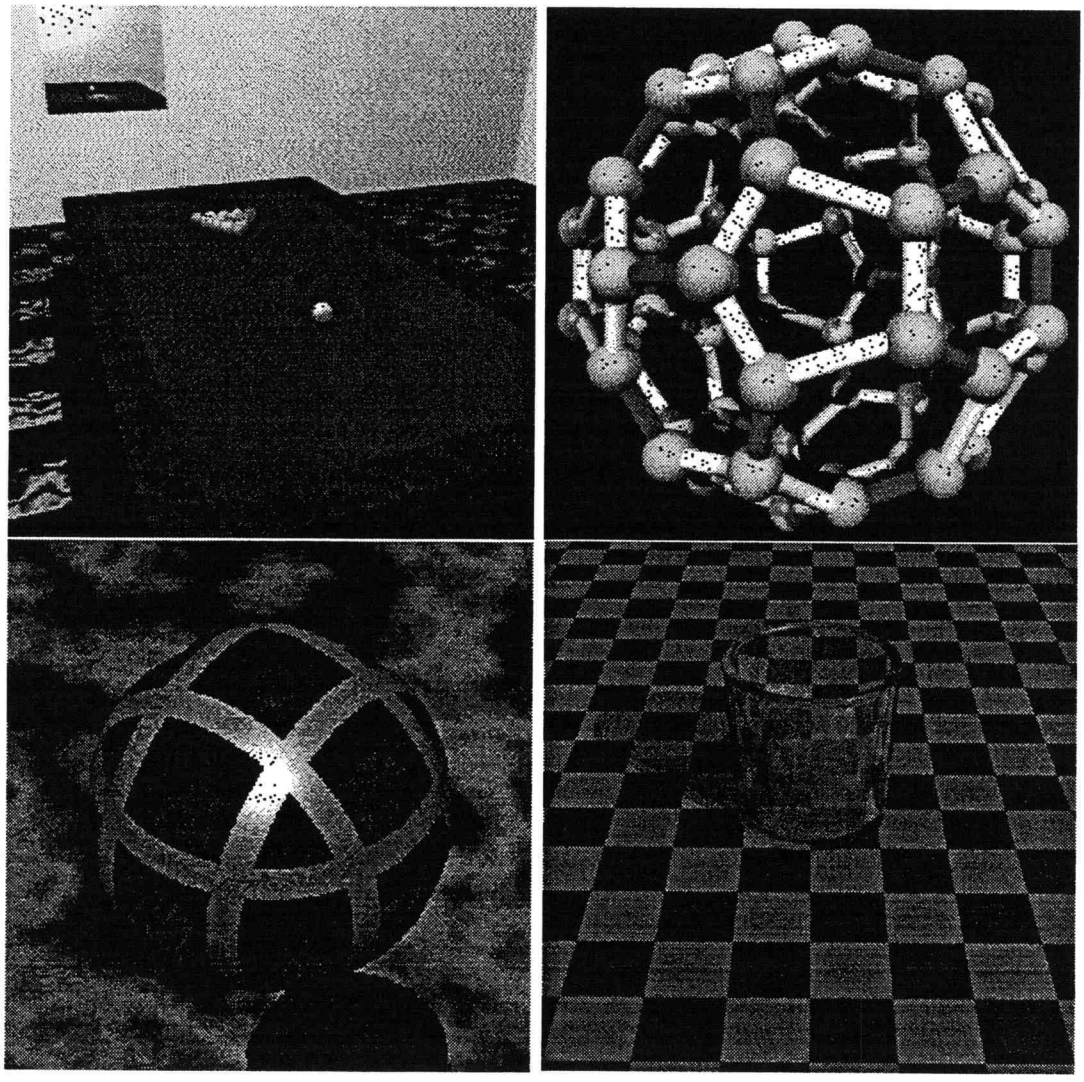

Figure 5.3. Sample images generated by the parallel raytracer

parallel implementation (Fig 5.3) . The parallel algorithm works by splitting the task into scanlines, each of which is computed by a *chare*. Since the computation involved varies from image to image, portion of the image, size of desired image and the maximum tracing depth, the time taken for each scanline will vary a lot. The scene to be raytraced is stored in a file which is parsed by *all* processors. To speed up the reading of the files, they are copied into individual `/tmp` directories so that all processors can perform their file reads in parallel. On completion of this, they start raytracing individual scanlines of the image.

The load balancing strategy automatically takes care of the distribution and migration of work. Once the scanline is computed, the resulting portion of the image is sent to the host, where it is displayed. The current algorithm is a two phase strategy with an initial coarse image being first developed, followed by a finer image in the next pass. The two pass strategy ensures that the user will receive relatively quick feedback on the image to be raytraced. The results obtained indicate that parallelization can yield good benefits and are included in this report.

5.1.3. All Pairs Shortest Path

The all pairs shortest path is based on the Dijkstra's single source shortest path algorithm and running it in parallel for every vertex. The completely connected weight matrix is replicated on all the processors and individual chares work on finding the single source shortest paths for the vertices assigned to it. Since the complexity of Dijkstra's single source algorithm is $O(n^2)$, the complexity of the all pairs shortest path is $O(n^3)$. Dijkstra's single source algorithm works by trying to find for a nonnegative weighted graph, the shortest path from a vertex v to all other vertices in the graph $G = (V, E, w)$. Each vertex has a cost associated with it (initialized to infinity) which is reduced on finding a cheaper path to the source

vertex v. For each iteration, the node with the cheapest cost is removed from the free list as it will be the cheapest way to reach v. Thus after each iteration, one node will be removed from the free list and the node from which the new reduced cost was found is marked as its predecessor. This process is continued until all nodes are removed from the free list.

5.1.4. 2-D Morphing

Morphing, a term derived from the Greek word *morphe* which means form or shape is a graphical process of altering one image to another through a series of intermediate images so that the viewer gets the impression that the first image has *metamorphised* into the second one. The term originated in the late 80's at George Lucas's Industrial Light and Magic (ILM). ILM developed a program called Morph which was made to to handle transformation scenes in the movie *Willow*. The morphing process varies depending on the technique used. Morphing algorithms can be classified as 2-D or 3-D. 2-D morphing gives the visual effect of a 3-D change of shape by warping a 2-D image from an initial shape to a final stage. Digital image warping algorithms are used to stretch and deform an initial image to the shape of the target. At the same time, textures for each image are gradually blended from the initial texture to the final one. To achieve greater control, the source and target images are broken up into small regions that map into each other. In 3-D Morphing, a 3-D geometric model of the object is transformed from one shape into another. At each stage of the metamorphosis, the 3-D model is rendered and texture mapped to produce a 2-D screen representation. 3-D morphing is considered to be difficult and and various problems occur when trying to morph 3-D objects that are structurally difficult like the case of morphing a torus onto a rectangle.

2-D Morphing, on the other hand, is simpler and is accomplished mainly through digital image warping and applying texture mapping [10]. Texture mapping by itself can be used to handle simple morphs. The basic technique in texture mapping is a two step process of mapping a 2-D texture plane on a 3-D surface and then projecting the surface on a 2-D screen display. Texture mapping serves to create an appearance of complexity by applying elaborate image detail to relatively simple surfaces. This mapping can be used to perturb surface normals and thus allow the simulation of bumps and wrinkles without the effort of modeling intricate 3-D geometries. Digital image warping leaves out intermediate steps of mapping to 3D object space and instead directly maps from one 2-D image (input image) to another 2-D image (output image).

Digital image warping algorithms use a variety of geometric transformations including affine, projective, bilinear and polynomial transformations. The equations for 2-D transforms use a 3x3 matrix multiplication algorithm based on homogeneous coordinates. Whenever image transformations are completed, "holes" and "overlaps" can occur. This results because the images are discrete and not continuous. Consequently pixels in one space do not always have to have an exact correspondence in another space. Deriving the alternate value in another space is done using interpolation and sampling. A host of interpolation algorithms exist including cubic, bilinear and cubic spline convolutions. The easiest approach is the nearest neighbor algorithm which is used in several algorithms [19].

To specify a morph, the animator specifies a correspondence between input and images. This is often performed using points, triangles or meshes. A public domain application called *morphine*, written by Adam Hall at Sun Microsystems, was

parallelized. Morphine uses triangular meshes, specified by the animator, to provide some degree of correspondence between the images. Given the input image, final image, the corresponding meshes on both and the degree of warping needed, any intermediate image of that triangle can be created. Creating these warped images while steadily increasing the degree of warping from 0 to 1 results in a morphing sequence (Fig 5.4).

The parallel implementation works by allotting several meshes of the image to participating processors and letting them work on parallel. On completion, they send their portion of the image to a display processor which collects and forms the resultant image and displays it on a window. To provide more parallelism, several future images are also computed simultaneously and stored as successive future images until all previous images are completed. However due to memory limitations and size of the image, this number of future images cannot be usually more than 30. To ensure that the images are shown in the right sequence, chares representing future work are not spawned until previous images have been displayed. This is accomplished by maintaining a list of pre-allocated images. Once an image is complete, it is displayed and the tasks for the next frame are spawned. Thus, there will always be enough space to store all frames being currently computed.

Figure 5.4. A sample morphing sequence

6. EVALUATION AND MEASUREMENT TECHNIQUES

6.1. PERFORMANCE EVALUATION

A good load sharing strategy should be adaptive and give good performance under varying conditions. It is very difficult to monitor the performance of different workstations simultaneously through messages and infer from them the performance degradations or improvements. A performance monitoring tool was written to display useful information reflecting the current and previous states of participating workstations. The graphs generated indicates whether a workstation was idling or doing computation and in addition, provides other information regarding its actual, current processing power. The success of a load distribution scheme is measured by considering the variance among the finish times of the participating workstations. The finish times mentioned throughout this report is the actual wall-clock time and not user time. This is because user times will vary widely among processors depending on load and processing capacity of machines and does not tell us anything about the effectiveness of the load sharing strategy. The wall-clock time mentioned in this report was measured using the `gettimeofday()` call. Generally, the greater the variance, the poorer the effectiveness of the strategy. Hence if the variance among finish times is very small in comparison to the total execution time, it can be inferred that the load distribution strategy was quite effective. To evaluate the performance of the load distribution strategy, it was run under normal load conditions on participating workstations. Then it was run under simulated medium and heavy load conditions and the ratio of the variance in finish times to the mean finish time is found out to determine the effectiveness of the strategy. Load was simulated on

participating workstations by running dummy CPU intensive processes. Chapter 7 deals with two components of the load distribution strategy, the dynamic scheduling phase and the load sharing phase. To determine the effectiveness of each of these components to the overall performance of the load distribution strategy, the components were disabled and selectively enabled. The time taken for each of the jobs under conditions of normal load was noted. This gives the reader a clear idea about the contributions of the two phases to overall performance increases.

6.2. PERFORMANCE MONITORING

Performance Monitoring was in-built as a feature of CHARM to enable the user to see how well the user was using the participating processors and how well the load balancing scheme was exploiting the available work. This performance monitor is not trace-driven and dynamically reflects the current status and previous history of events that have occurred on the processors (Fig 6.1). The display is scalable and extensible. Primarily, the window shows the time spent on computation, initialization of data, idling and the number of tasks processed.

The performance monitoring window is created and maintained on a CHARM host only when Charm is run in a performance monitoring mode. In this mode, the workstations which take part in the parallel computations periodically send status messages to the host explaining what they are working on now and how many tasks have been processed currently. These periodic reports are analyzed at the host and converted to appropriate graphical information which is then displayed to indicate the progress made. Since wall-clock time is used to calculate all times, the display is not exactly accurate as the workstation could be working on other user tasks while it is indicated to be doing useful computation or idling on the performance monitoring

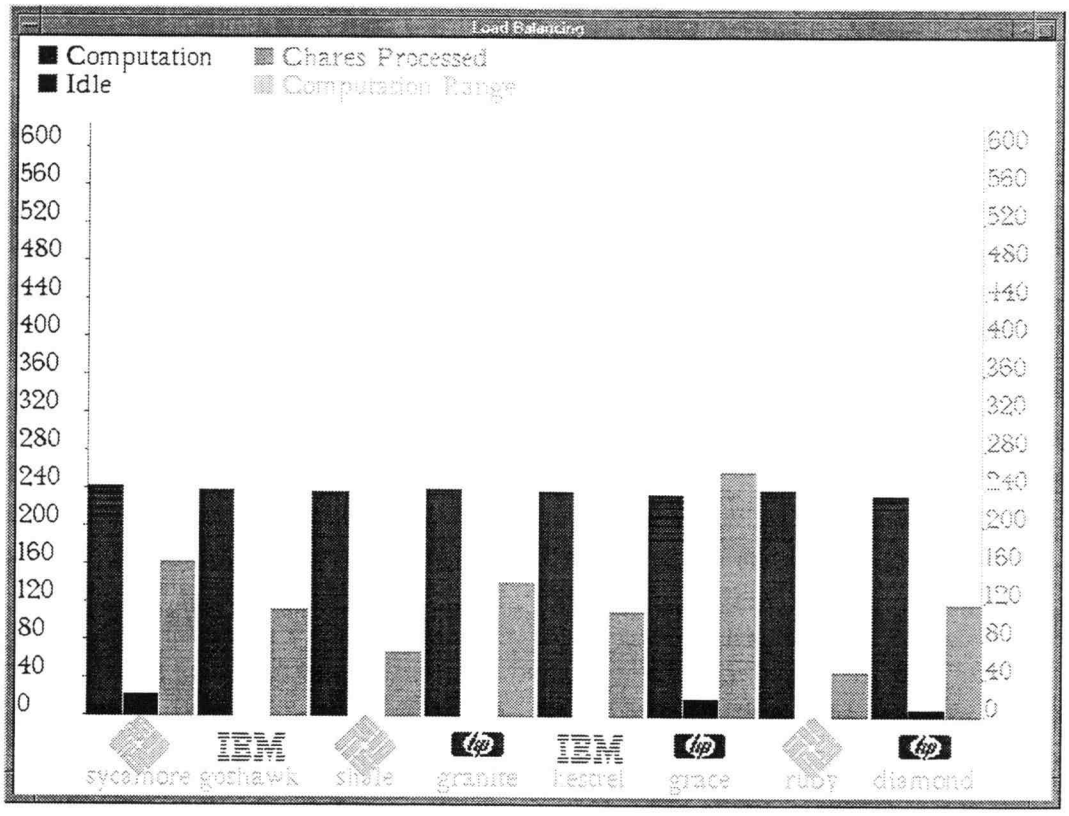

Figure 6.1. Performance monitoring

window. However, it is very easy to spot general trends and these trends remain fairly consistent irrespective of the type of clock used to measure time.

When Charm is run with the performance evaluation mode turned on, the status and history of each processor is maintained by a record of the total time spent, total time spent idling, and the number of tasks completed. The display is cumulative and a quick look at the bar charts obtained indicates when processors were idling and how much they were idling. The ratio of total time spent to chares processed gives an idea of the actual processing capacity of the participating workstations. The performance window can be extremely useful to detect load imbalances quickly and execute corrective action, either by rewriting the application program or by improving initial scheduling.

6.3. LOAD TRANSFER MONITORING

It is often difficult for the application programmer to figure out why the load balancing algorithm worked the way it did or did not perform according to the user's expectations. To help understand better how the load balancing works, an animation window has been developed which can be brought up while the user is running the application program (Fig 6.2). This window shows the available processors and the peer connections between them. Requests for work are represented by empty wheelbarrows, work transfer by loaded wheelbarrows, and work request rejects by red empty wheelbarrows. Fig 6.2 shows two instances of one of the animations, the first one shows a snapshot of the animation while all workstations have enough work and the second one shows an instance towards the end of the computations.

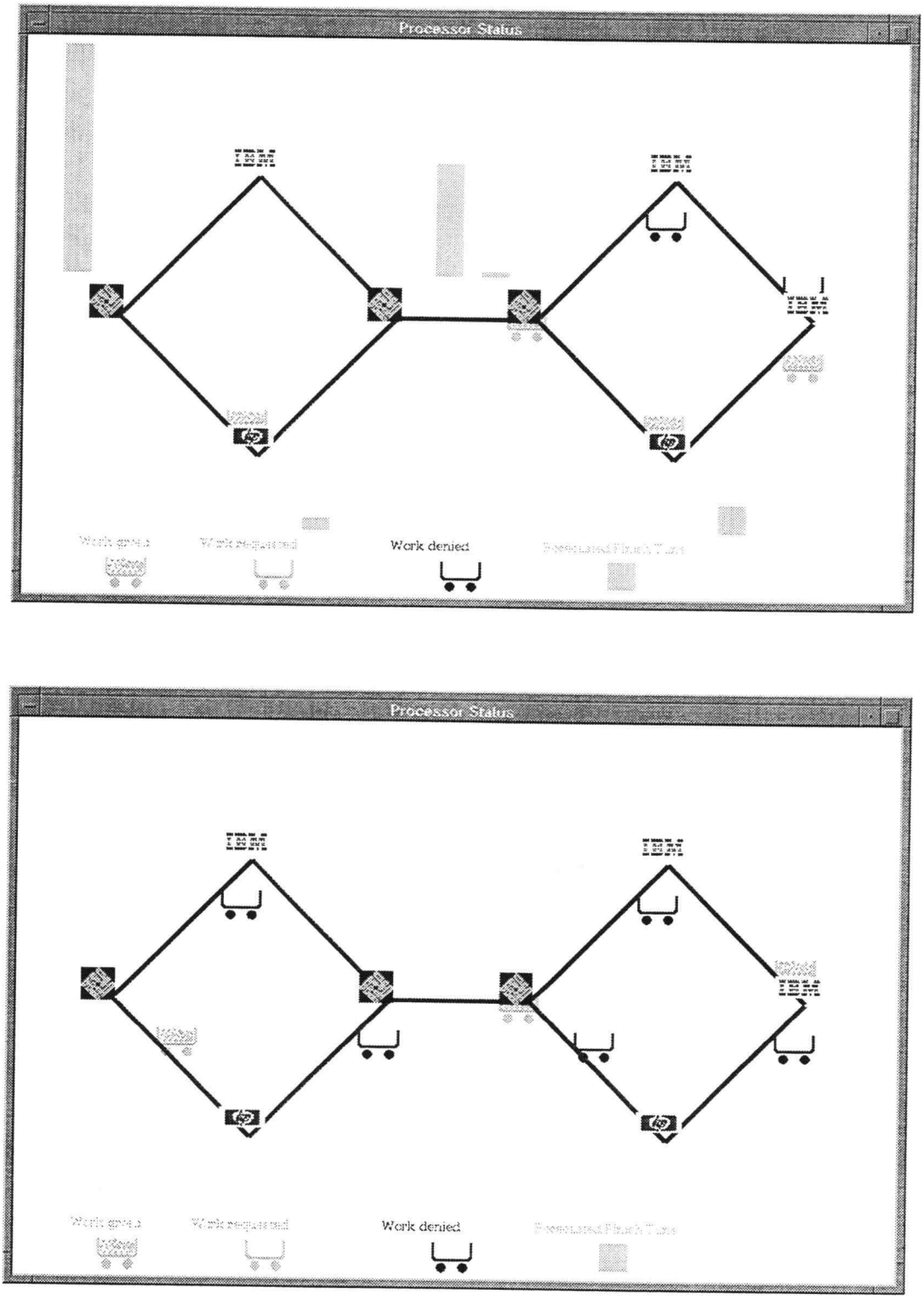

Figure 6.2. Load transfer monitoring

During the actual animation, the wheelbarrows scuttle back and forth from their processors, requesting and transferring work. The Forecasted Finish Time (FFT) associated with each workstation is updated constantly to show how much of the estimated work is left on each processor. The peer connections between workstations flash when a request for work, work transfer, or work denial occurs, alerting the user's attention. When a processor requests for work, an empty wheelbarrow travels towards the workstation from which work is requested from. If the workstation decides to transfer work, the wheelbarrow comes back full of work. However, if the workstation denies the request for more work, the wheelbarrow is returned back empty and its color is changed to red to indicate that the processor was rejecting an offer of transferring work and not itself requesting for more work.

Initially the FFT bars will be quite high indicating that all workstations have enough work. As the animation proceeds, the stock of available work diminishes and the workstation begin asking their neighbors for work. When work gets transferred, the FFT bars are updated accordingly to show the new distribution of the total work. Towards the end of the computations, the dynamic load distribution strategy gets activated, indicated by a flurry of work transfer and work request activity. Finally, each node quits trying when the retry attempts have been exceeded or when no potential neighbor with work can be found.

7. APPLICATION PROGRAM RESULTS

7.1. EVALUATION METRICS

Test results were obtained by running three different variations of the load distribution strategy under conditions of normal load, mild or no load, and when some of the workstations had moderate to heavy loads. Mild or no load conditions were simulated by running the applications at night when user activity was less than normal and moderate to heavy load conditions by giving some of the workstations extra work to process. The variance of the finish times was then examined to find out how effectively the load sharing strategies performed.

The performance of different load distribution strategies can be evaluated by the total time taken for the parallel job and the total time idled for lack of work. Since the workstations were never available for dedicated use, the total time taken often varies depending on the current workload on individual workstations. Therefore the variance of the finish times, along with the total parallel time taken, provides us with a better measure of the performance of the load distribution strategies. However, even the variance by itself does not give us enough information when dealing with multiple programs of differing execution times. To test the efficiency of a load sharing strategy, the standard deviation was expressed as percentage of the mean of the individual computation times. For a good load distribution strategy, this number should be fairly constant and never be large. A large value would indicate that the load distribution strategy did a poor job of scheduling and transfer of data.

When comparing results, it must be kept in mind that the results were obtained in a multi-user environment and hence variations in the parallel times taken are possible due to increased or decreased user activity. It must also be understood that this report places more emphasis on how well the load has been balanced or shared among participating workstations and not so much on the performance increases obtained, although better load distribution indirectly ensures higher speedups.

Four sample programs were taken (Refer Chapter 5) and used as the testbed to evaluate the load sharing strategy. Since some of the applications are fairly complex and differ in functionality from their sequential versions, all timings and hence performance increases have been calculated based on timings obtained when the parallel program was run in a uni-processor mode. Care was taken to see that this would not bring any additional performance degradations except those caused by the Charm run-time system.

To fully test the different components of the load distribution strategy and their contribution to the overall results, different parts of the load distribution strategy were turned off and individual components selectively turned on. Then the suite of test programs was run with just these components turned on and the performance results found out. The load distribution strategy is basically composed of two main parts, the dynamic scheduling part and the task transfer part. In the experiment conducted, three versions of the load sharing strategy were tested.

- **Run-time Scheduling Strategy:** Work is distributed based on processing capacities. The load distribution strategy has the work transfer component, status component, location and retry component, all switched *off*.

- **Dynamic Work Transfer Strategy:** The load distribution strategy with with initial task distribution performed *equally* irrespective of machine loads and processing capacities. Dynamic transfer of work is however enabled and performed when individual workstation are about to run out of work.

- **Combined Strategy:** The load distribution strategy where both scheduling and work transfer components are enabled and functioning. In this strategy, tasks are initially distributed based on processing capacities and work transfers are later performed if necessary when individual workstations idle for lack of work.

The performance monitoring tool was turned on while the three strategies were run and the results (Figures 7.1, 7.2 and 7.3) indicate the performance characteristics obtained while running the raytracing application using the Run-time Scheduling Strategy, Dynamic Work Transfer Strategy and Combined Strategy respectively. The image raytraced while obtaining the graphs was a 800×800 pixel image of a buckminsterfullerene molecule. When the Run-time Scheduling Strategy was used, (Figure 7.1) the variance was maximum. This is because no dynamic migration of work is performed. The Dynamic Work Transfer Strategy (Figure 7.2) has a much lower variance among the finish times but the total time taken is more, compared to the Combined Strategy (Figure 7.3). This is because of the additional overhead involved in the migration of tasks. The Combined Strategy delivers good performances because the initial scheduling ensures that the work transfer strategy does not have to work very hard to balance the workload. This is indicated by the low variance among the finish times of individual workstations and a minimum of idle time.

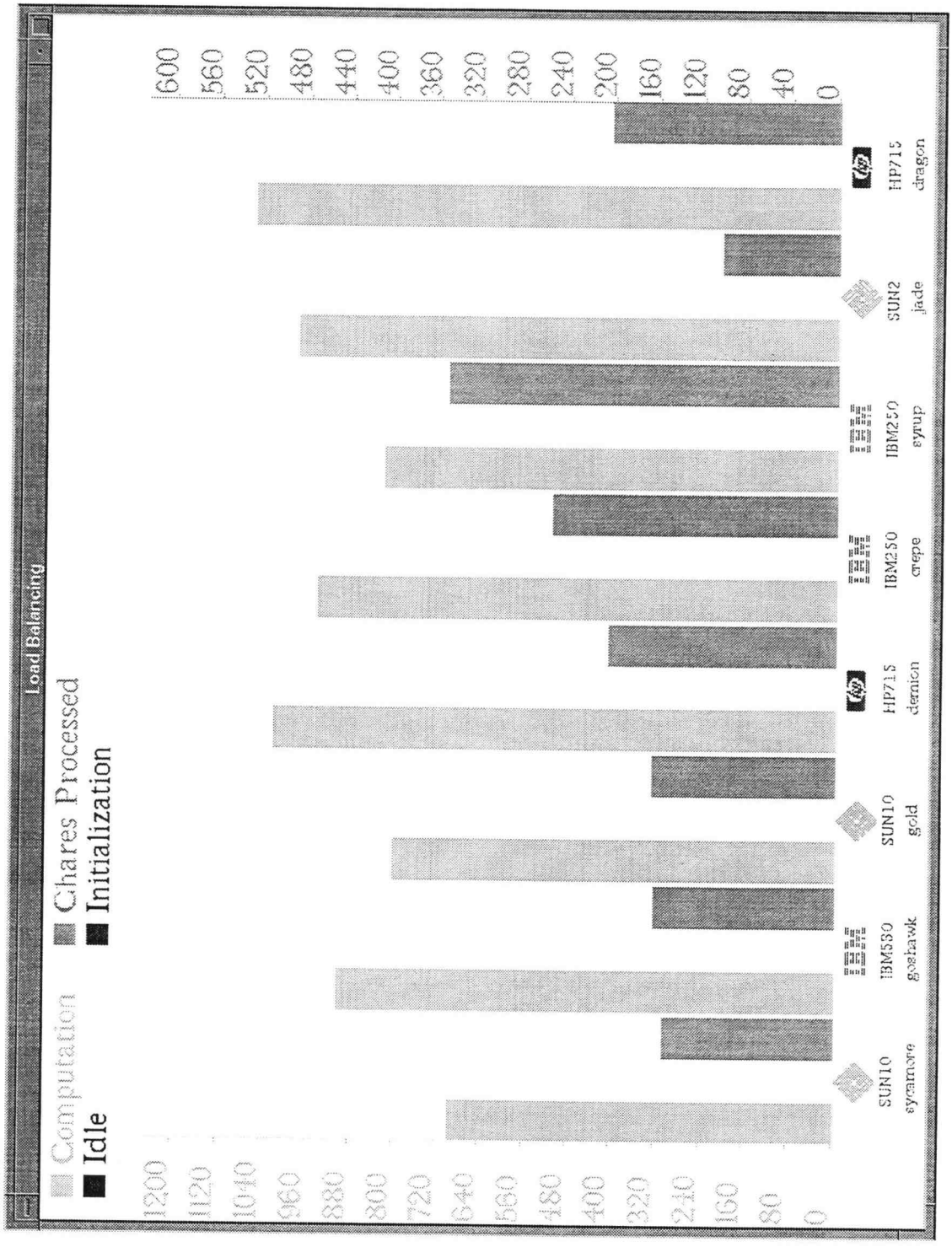

Figure 7.1. Performance results for the run-time scheduling strategy

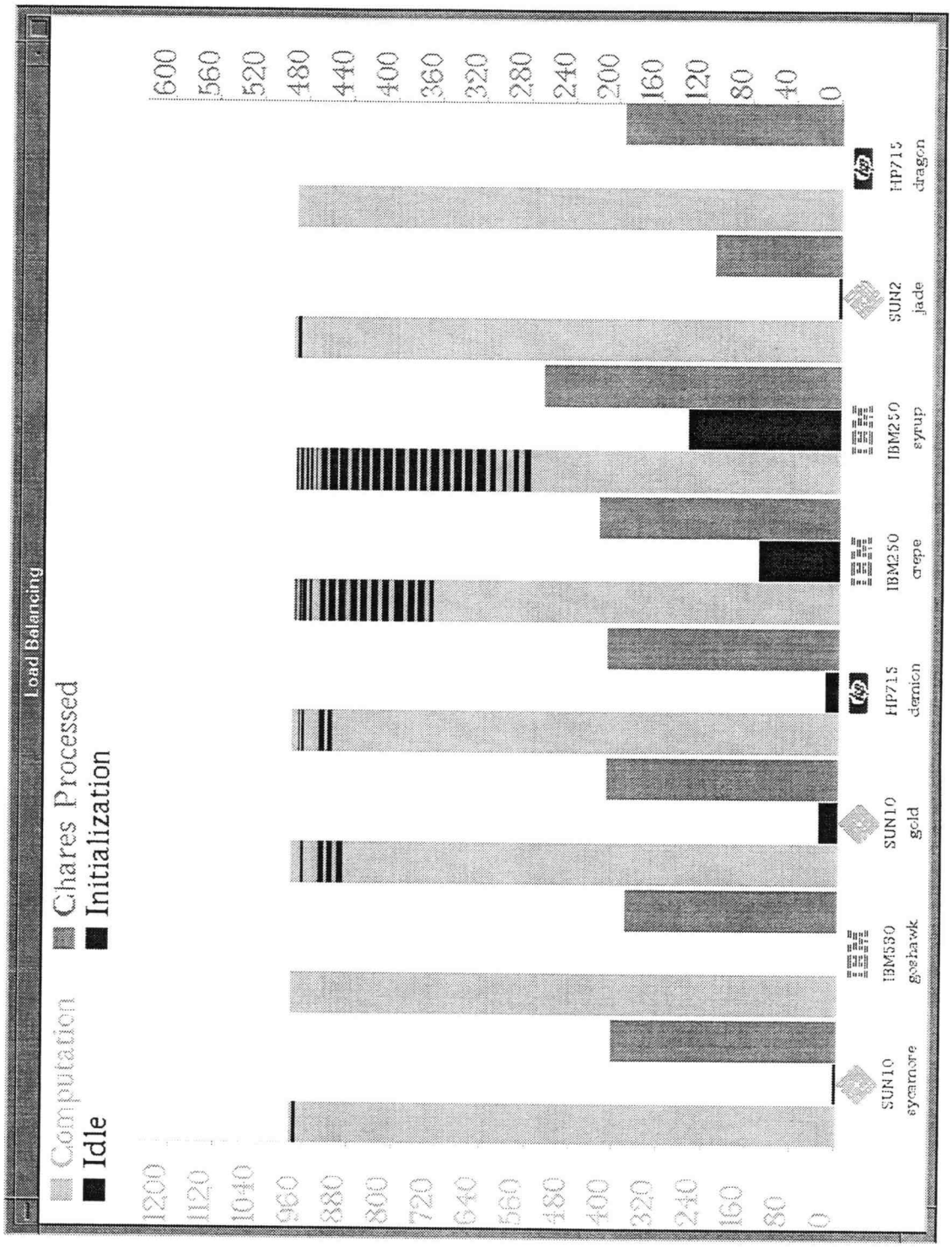

Figure 7.2. Performance results for the dynamic work transfer strategy

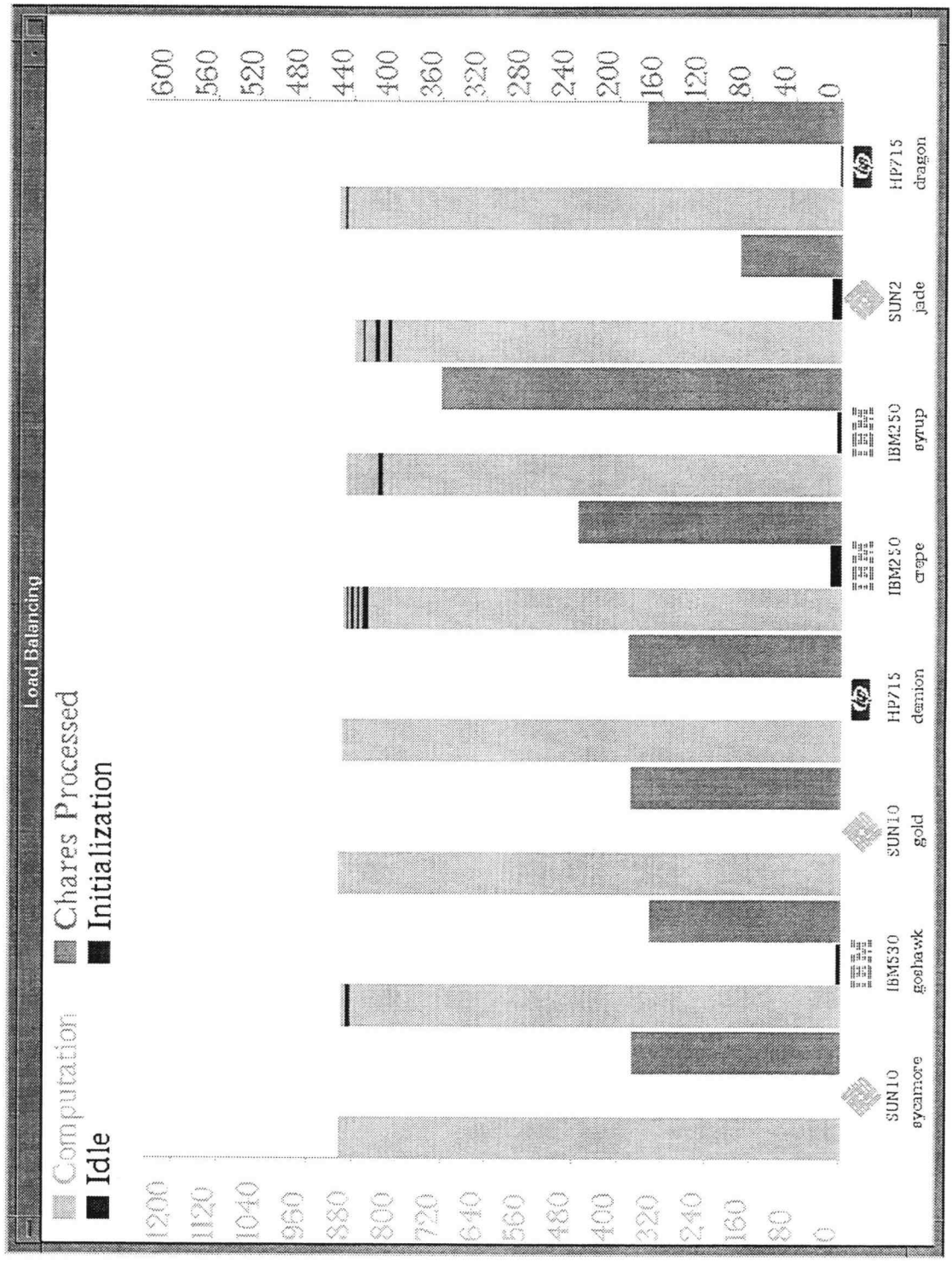

Figure 7.3. Performance results for the combined strategy

7.2. TEST RESULTS

The four sample programs were run with these strategies and the results obtained are shown in Tables 7.1 and 7.2. Each value obtained is the average of ten program executions under suitable load conditions. These results provide us with useful hints as to the effectiveness of the different components to the overall load sharing strategy. Initially, the different components of the load distribution strategy were individually enabled under normal conditions. Later, the effectiveness of the combined strategy was demonstrated by running it under varying conditions of load and by measuring the responsiveness of the strategy in coping with load imbalances.

7.2.1. Performance Under Normal Computational Loads

The run-time scheduling strategy, fared the worst because of its comparatively un-adaptive nature. This strategy has no way of coping with load fluctuations after the program has begun execution. This becomes specially apparent when the application runs for a long time, like the raytracing application, and thereby being more susceptible to load imbalances. The Run-time Scheduling strategy performs best when the amount of data associated with the task is high with respect to the amount of computation . If the work transfer strategy has to migrate large amounts of work to counter heavy load imbalances, the costs associated with work migration and the high message latencies on the Ethernet rapidly deplete any performance advantages gained by the migration. The scheduling strategy also delivers good performances for algorithms when there are a limited number of moderately computation intensive tasks to be distributed followed by communications or a synchronization. This is specially true for graphical algorithms were the problem size cannot be scaled

up to increase the number of tasks per communication or synchronization because the dynamic strategy is not nimble enough in an Ethernet environment to migrate work, in response to small workload fluctuations. The 2 D Morphing application is an example. Here, only a small number of consecutive images can be completed in parallel (about 30 consecutive future images) at a time. Successive images can be processed only on completion of earlier images due to memory limitations.

The dynamic work transfer strategy performs highly effectively when there are a large number of free tasks with low amounts of data associated with them. This is why it performs so well for the All Pairs Shortest Path and Raytracing applications (Table 7.1). However its efficiency deteriorates when the amount of data associated with the task increases and when the amount of computation per communication is small. In spite of inefficiencies for such special applications, the over-all performance for the strategy is quite high. Even for the worst case, the ratio of standard deviation to mean computation time is still less than 6%. Thus the dynamic work transfer strategy is highly recommended if the user wants fairly decent performance increases and does not want to take the trouble of running scaled down problems on each individual workstation and determining the processing capacity of the workstation for that application.

The combined strategy performs best because it is able to utilize the good features of both the scheduling and the dynamic work transfer strategies. The overall standard deviation to mean computation time ratio is very low. The maximum recorded ratio was less than 2.5% for the four applications, with the average a meager and extremely healthy 1.5%. This indicates that the combined load sharing strategy is very adaptive to most of the application programs and delivers good

Strategy	*Application Programs*	*Total* *	*Mean* *	*Std Dev* *	*% Std Dev*
Run-time Scheduling	All Pairs (1500 vertices)	265.7	219.4	30.97	14.12
	Matrix Mply (1200×1200)	154.0	135.7	16.81	12.39
	Raytrace (800×800 pixels)	1412.8	982.46	201.9	20.55
	2 D Morphing (250 steps)	279.6	267.8	4.97	1.86
Dynamic Work Transfer	All Pairs (1500 vertices)	236.5	234.1	1.85	0.79
	Matrix Mply (1200×1200)	138.4	133.3	3.03	2.27
	Raytrace (800×800 pixels)	1248.5	1232.2	10.7	0.87
	2 D Morphing (250 steps)	326.4	297.0	17.65	5.94
Combined	All Pairs (1500 vertices)	214.9	212.7	1.83	0.86
	Matrix Mply (1200×1200)	133.3	131.0	1.87	1.43
	Raytrace (800×800 pixels)	1009.9	981.0	20.69	2.1
	2 D Morphing (250 steps)	282.8	272.3	4.4	1.61

Table 7.1. Efficiency of load sharing strategies under normal loads

all-round performance. Hence, the combined strategy can be used to deliver very good results when a task needs to be run repetitively a large number of times. This would justify running it sequentially on the available workstations and determining their relative processing capacities.

---

*All values of time are in seconds

	Low Loads			*High Loads*		
Programs	*Mean**	*Std Dev**	*% Std Dev*	*Mean**	*Std Dev**	*% Std Dev*
All Pairs (1024)	72.24	0.57	0.78	94.22	1.56	1.65
Matrix (1200×1200)	129.71	2.00	1.5	238.42	16.51	6.92
Raytrace (800×800)	901.39	8.16	0.91	1378.16	17.10	1.24
Morphing (200 steps)	227.06	9.04	3.98	393.25	7.43	1.89

Table 7.2. Performance changes under varying conditions of load

7.2.2. Performance Under Varying Computational Loads

The previous section has demonstrated the effectiveness of the load sharing strategy under conditions of normal load. To demonstrate the adaptivenes of the load distribution strategy, the combined strategy was run on machines, under conditions of both high over-all load and low over-all loads. Load was simulated by running *several* (about 2-4) dummy *for* loops on several of the faster machines. For example, the first, second and fourth workstations were subjected to heavy load. The load distribution strategy performed fairly well despite the slow response of the some of the machines to status updation and work transfer messages. The average standard deviation to mean computation time ratio increased only to 2.92% from 1.11%. The ratio actually *decreased* for the 2 D Morphing application as the heavy load on some machines actually reduced the processing capacity disparities among the faster and slower workstations, thereby allowing the processors to finish together, despite the increase in overall total execution time.

7.3. HETEROGENEOUS SPEEDUPS

Although the primary emphasis of this article is on efficient load distribution, a good distribution is meaningless without any performance increases. Conventional notions of speedups are not well suited for heterogeneous environment. This is because the processing capacities for different machines vary, according to the number of jobs currently running and their respective processing capabilities. Hence, it is not immediately apparent which sequential time should be taken to compare with the parallel time to obtain the speedups. Taking the slowest machine of the network will result in exaggerated speedups while taking the fastest machine for comparison will yield very conservative speedups. Hence a better method of calculating speedups is needed on a heterogeneous network so that the meaning of speedups is still conserved. The method used in this report is to consider all differing participating workstations and and rate them relative to the fastest processor. This means that the application program is run on all the machines and the uni-processor time obtained is used to rate the machines. Once these relative rates are obtained, the total computing power present in the network can be represented as being equivalent to having say x of the fastest processors (Table 7.3). If there are n heterogeneous processors, then $x \leq n$.

In a typical homogeneous environment, an ideal speedup curve will be linear. In a heterogeneous environment this curve will be sub-linear provided it was calculated with respect to the fastest workstation. Individual points on the curve are calculated by usual methods, except that the sequential time on the *fastest* pro-

---

*All values of time are in seconds

Uni-processor Timings *					
Programs	*SUN 2*	*SUN 10*	*HP 715*	*IBM 530*	*IBM 250*
All Pairs	880.1	492.3	710.5	765.5	303.1
Matrix Mply	2349.5	1484.0	1315.0	961.2	576.6
Raytracing	15400.3	8351.6	7686.6	8582.5	4271.9
2D Morphing	2171.5	1074.6	1894.7	1022.2	570.9
Relative Processing Capacities					
Programs	*SUN 2*	*SUN 10*	*HP 715*	*IBM 530*	*IBM 250*
All Pairs	0.34	0.62	0.43	0.40	1.00
Matrix Mply	0.25	0.39	0.44	0.60	1.00
Raytracing	0.28	0.51	0.56	0.43	1.00
2D Morphing	0.26	0.53	0.31	0.56	1.00

Table 7.3. Relative ratings of workstations for application programs

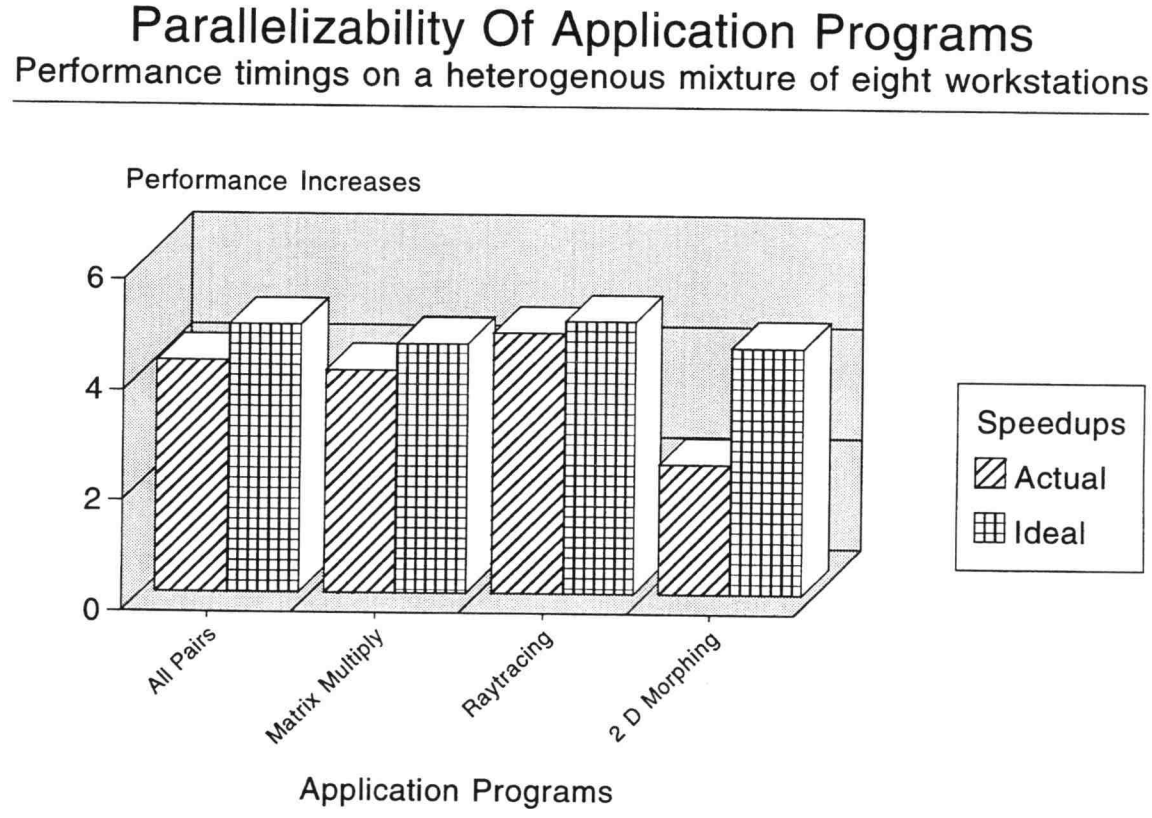

Figure 7.4. Ideal and actual speedups for the test suite

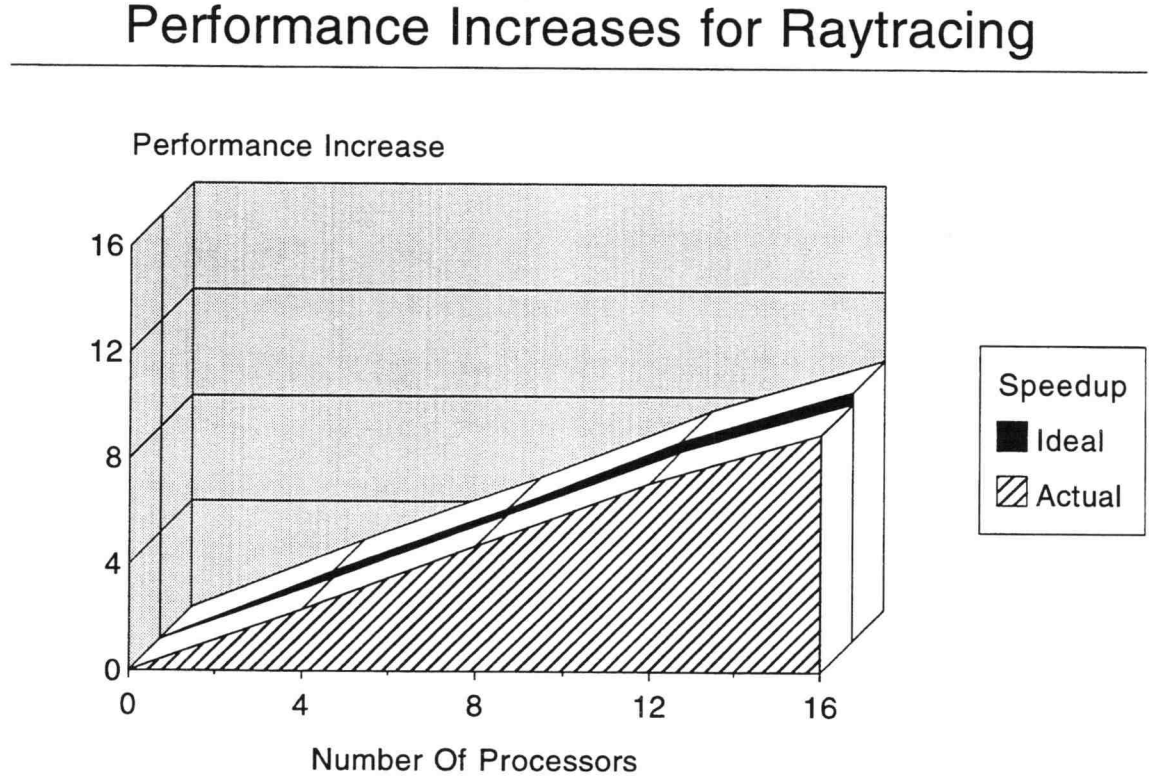

Figure 7.5. Variation of speedups with increasing number of workstations

Parallel Time *	*Ideal Speedups*	*Actual Speedups*	*Percentage Utilization*	*Total Workstations*
1838.9	2.57	2.32	0.90	4
906.8	4.92	4.71	0.96	8
601.2	7.49	7.11	0.95	12
476.7	9.40	8.96	0.95	16

Table 7.4. Utilization of available computing power for raytracing

cessor is divided by the total parallel time to obtain the speedup. The ideal or maximum possible heterogeneous speedup is determined by plotting x on the graph. Figures 7.4 and 7.5 illustrate these concepts better. The results indicate that the application programs deliver extremely good performance increases on being run in parallel, especially since the workstations were never available for dedicated use.

Since the actual shapes may vary for the ideal and actual speedup curves depending on the sequential time used for comparison, the percentage of total available computing power actually used, can be used as a good measure of how well the distribution strategy was able to distribute work along with the speedup curves. To measure this, the ratio of the actual to ideal speedups is taken. Sample values taken for the raytracing application (Table 7.4) indicate that the utilization is quite satisfactory even while the number of workstations was varied from 4 to 16 (Fig 7.5). For the raytracing application, the average utilization was 94.05%.

Workstations					
SUN 2	*SUN 10*	*IBM 530*	*HP 715*	*IBM 250*	*Total*
0	1	1	1	1	4
1	2	1	2	2	8
1	3	2	3	3	12
2	4	2	5	3	16

Table 7.5. Heterogeneous workstation cluster used for raytracing

*All values of time are in seconds

8. SUMMARY

8.1. CONCLUSIONS

Current trends indicate that parallel computing on workstation clusters is a very viable area for future research and work. In such an environment, especially when computers from various vendors are present, run-time environments like Charm can provide an easy interface for efficiently utilizing all the available resources. Since most of such environments are not available for dedicated use, some form of scheduling or load balancing is imperative for performance gains. This problem is general enough and too tedious to be the sole responsibility of the applications programmer. System modules which automatically schedule and distribute load, help in encapsulating and isolating these issues so that the programmer is not hampered by issues not directly related to the problem.

8.1.1. Recommendations

The results obtained from this work indicate that excellent benefits can be obtained by using the load distribution strategies for various classes of problems. The dynamic work transfer strategy, where work is equally scheduled in the beginning, is sufficient to deliver fairly good performance increases compared to absolutely no load sharing at all. This strategy can be used when the parallel application is used only infrequently. However, if better performance increases are needed and the application is run quite frequently, it might make sense to use the combined strategy instead. This strategy delivered consistently good performances for almost all the

test applications. When using the combined strategy, several levels of performance increases can be obtained depending on the amount of time the application programmer is prepared to spend improving performance. One can use SPEC_INT92 ratings, SPEC_FP92 ratings, or a mixture of both of them to gradually increase the savings. Finally, scaled down versions of the problem can be run in a uni-processor mode to best estimate the actual processing capacity of each workstation type.

8.1.2. Shortfalls

The scheduling and load distribution modules do a very good job of ensuring that all the participating workstations finish the available work at about the same time. However it should not be treated as a panacea for all problems that heterogeneous computing brings. Additions and modifications are still possible to make the strategy more efficient. One severe shortfall of the strategy lies in its notion of how data will be distributed. Since the strategy is two pronged, with an initial central distribution followed by a distributed approach to work migration, application programs have to be written to conform to this paradigm to obtain good speedups. Although this is considerably less work to do than writing specific load sharing modules for each parallel Charm application program, it sometimes does not work so well for problems which do not conform to such a paradigm and undermines the basic arguments proposed in this thesis for program independent load sharing and distribution. Perhaps the solution to this problem is to have several of such load sharing modules each of which is built with a specific class of programs in mind. Typically, the scheduling portion of this algorithm is useful for graphical applications where the problem sizes can be easily predetermined at start of computation. Scheduling hints, though contrary to the spirit of application program transparent

load distribution, can ease some of the problems by providing directives to the load distribution strategy regarding the nature of the parallel program.

8.2. FUTURE WORK

Although heterogeneous computing raises a number of unique and challenging load sharing issues, several areas of intelligent scheduling and dynamic load sharing remain unexplored. Currently, the Charm run-time system works on SunOS, HPUX and AIX operating systems. Work is going on to port it over to IRIX, OSF/1 and other operating systems for greater heterogeneity. There is also a need for a operating system independent protocol for data transmission. Formats such as XDR can be used to achieve this.

8.2.1. Programming Hints for Intelligent Transfer

Unfortunately, experience has shown that performance can always be improved with the help of programmer issued hints and directives. Although such principles are counter to the spirit of programmer independent scheduling and load sharing, better performance results can be obtained through very simple programmer directives. Hence, it is felt that the programmer should always be given the option to improve performance if he is unhappy with current performance levels. Thus a two tier scheme of improving performance can be used. During the first level, Charm is allowed to have its way regarding task scheduling and migration. If the programmer feels there is scope for improvement, additional directives are placed in the program. The scheduling hints mentioned in Chapter 4 are good examples of such directives. A stumbling block that still exists is the variable nature of work in each chare. Currently, the predictions are calculated by finding out the number of tasks processed in the last t seconds. More sophisticated strategies which take

into account the differing grainsize of various tasks can be utilized if the system is given some directives regarding the nature of work associated with free chares in the work queue. Simple directives regarding the computational complexity associated with a chare can give the the system a better idea whether the costs of migration outweighs the time saved by doing it in another processor. Thus each chare can be considered individually on its merit to be migrated or executed locally. Learning algorithms can also be used which remember the time taken by a chare doing similar work earlier, and use that information in future transfers of identical chares.

8.2.2. Multiple Networks

The existing Charm environment works only on the Ethernet. Interesting possibilities emerge when some of the workstations are connected by Ethernet, others by ATM, FDDI or other specialized high speed network. Then, work migrations are cost effective only when the underlying network connection is considered by the work transfer policy. Task migrations are decided based not only on the nature of work to be transferred but also the time taken to transfer it in the underlying network. If more than one network connection is possible between two workstation, the Charm system should automatically use the fastest alternative available. The decision of which network to use should be program transparent. This implies that the programmer should never need to know how workstations communicate with each other except that they'll use the fastest means possible.

8.2.3. Metacomputers

The differences between a high speed dedicated workstation network and a parallel computer are rapidly dissolving. With faster networks like ATM, vector processors, parallel computers and workstation clusters can be viewed as computing entities or metacomputers. Several of these metacomputers can be connected to each other and dynamically exchange tasks and status information. This entails a multilevel view where individual computers in a metacomputer exchange information and data among themselves. Metacomputers, similarly exchange information and work when all individual machines comprising it have run out of work or are about to run out in the near future.

Some work in this regard has already been completed using the 16 processor Meiko Supercomputer and a eight node heterogeneous workstation cluster. Initial results have been encouraging. The individual nodes in the parallel machine communicate with each other using a fast communication network and with the other workstations using Ethernet. The Charm environment automatically decides which communication protocol to use depending on the underlying network connection.

BIBLIOGRAPHY

[1] O. Babaoglu, L. Alvisi, A. Amoroso, R. Davoli and L.A. Giachini
Run-time Support for Dynamic Load Balancing and Debugging in Paralex, TR 91-1251, Department of Computer Science, Cornell University, December 1991.

[2] A. Beguelin, E. Seligman and M. Starkey
Dome: Distributed object migration environment, Technical Report CMU-CS-94-153, School of Computer Science, Carnegie Mellon University, May 1994.

[3] F. Douglis and J.K. Ousterhout
Transparent Process Migration: Design Alternatives and the Sprite Implementation, Software –Practice and Experience, Vol 21, No. 8 Aug 1991 , pp 757-785.

[4] D.L Eager, E.D Lazowska and J. Zahorjan
A Comparison Of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing, Performance Evaluation, Vol 6, No 1, pp. 53-68 March 1986.

[5] D.L Eager, E.D Lazowska and J. Zahorjan
Adaptive Load Sharing in Homogeneous Distributed Systems, IEEE Transactions on Software Engineering, May 1986, pp. 662-675.

[6] W Fenton B. Ramkumar, V. Saletore, A. Sinha and L.V. Kale
Supporting Machine Independent Programming on Diverse Parallel Architectures, International Conference on Parallel Processing, St. Charles, Ill., August 1991, pp. 193-201.

[7] J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes
Computer Graphics: Principles and Practice, 2nd Ed. Addison-Wesley 1990

[8] A. Geist, A. Beguelin, J.J Dongarra, W. Jiang, R. Manchek and V.S Sunderam
PVM 3 User's Guide and Reference Manual, Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[9] A.S. Grimshaw, J.B. Weissman and W.T. Strayer
Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing, Technical Report No. CS-93-40 July 14, 1993.

[10] Valerie Hall
Morphing in 2-D and 3-D, Dr Dobb's Journal, July 1993, pp. 18-26.

[11] L.V Kale
The Chare Kernel Parallel Programming Language and System, Proceedings of

the International Conference on Parallel Processing, August 1990, Vol. II, pp. 17-25.

[12] P. Krueger and R. Chawla
The Stealth Distributed Scheduler, Proc. 11th Intl Conf. Distributed Computing Systems, IEEE CS Press. Los Alamitos, Calif, 1991, pp. 336-343.

[13] Thomas Kunz
The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme, IEEE Transactions On Software Engineering, Vol 17, No 7, July 1991.

[14] M.J Litzkow, M. Livny and M.W Mutka
Condor - A Hunter of Idle Workstations, Proc. Eighth Intl Conf. Distributed Computing Systems, IEEE CS Press, Los Alamitos, Calif. 1988. pp 104-111.

[15] N. Nedeljkovic and M. J. Quinn
Data-Parallel Programming on a Network of Heterogeneous Workstations, Concurrency: Practice and Experience 5,4 (June 1993), pp. 257-268.

[16] Cornel K. Pokorny & Curtis F. Gerald
Computer Graphics: The Principles Behind the Art and Science, Franklin Beedle & Associates 1988.

[17] Niranjan G. Shivaratri, Phillip Krueger and Mukesh Singhal
Load Distributing for Locally Distributed Systems, IEEE Computer, Vol 25, No 12, December 1992.

[18] W. W. Shu and L. V. Kale
A Dynamic Load Balancing Strategy for the Chare Kernel System, Proc. of Supercomputing '89, November 1989, pp. 389-398.

[19] G. Wolberg
Digital Image Warping, IEEE Computer Society Press, Los Alamitos CA, 1990.