ADAPTIVE PARALLEL COMPUTATIONS ON NETWORKS OF WORKSTATIONS

By

Mohan V. Nibhanupudi

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

Dr. Boleslaw K. Szymanski, Thesis Adviser

Dr. Thomas E. Cheatham, Jr, Member

Dr. Mukkai S. Krishnamoorthy, Member

Dr. Franklin T. Luk, Member

Dr. David R. Musser, Member

Rensselaer Polytechnic Institute
Troy, New York

April 1998
(For Graduation May 1998)

ADAPTIVE PARALLEL COMPUTATIONS ON NETWORKS OF WORKSTATIONS

By

Mohan V. Nibhanupudi

An Abstract of a Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Rensselaer Polytechnic Institute
Troy, New York

April 1998
(For Graduation May 1998)

To my wife, Padma

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

A Ph.D. degree is a life experience unlike any other. It requires far more effort, drive and determination on the part of the individual and calls for emotional support from teachers, guides, friends and family. A number of individuals helped me in the pursuit of my goals. I acknowledge the contributions of each and every individual who directly or indirectly contributed to my learning and well being.

First and foremost, I would like to thank my advisor, Prof. Boleslaw Szymanski, for his constant encouragement and support throughout my Ph.D. program. I am especially thankful for his guidance and helpful pointers at various stages of my Ph.D. His guidance helped me to focus on the more important aspects of my research. I also improved my style of writing under his guidance.

Prof. David Musser, as my advisor during the first year of my graduate studies at Rensselaer, helped me in my transition to Ph.D. level research. I thank him for his support and guidance throughout my Ph.D. program. I thank Prof. Mukkai Krishnamoorthy for his advice and support during the course of my thesis. I am thankful to Prof. Thomas Cheatham Jr., and Prof. Franklin Luk for their valuable suggestions. My thanks are also due to Prof. Cheatham for being available for my defense at such short notice. I thank Prof. Mark Goldberg for his help at a crucial stage in my thesis.

I have had the opportunity to work with a number of my friends: Charles Norton, Bill Maniatty, Toshiro Ohsumi, Wes Turner, and Dave Hollinger. I thoroughly enjoyed working with them and learned a great deal in the process. A number of friends helped ease the moments of pressure and make my stay here a pleasant experience. I acknowledge the support of Ray Loy, Kishore Bubna, Louis Ziantz, Chak Bhagvati, John Punin, Ugur Dogrusoz, Badri Ramamoorthy, Kedar Tupil, Ewa Deelman and several others. The “lunch group” provided the necessary distraction and a pleasant lunch experience.

I thank the lab staff who work hard to maintain the CS network. My special thanks are due to Nathan Schimke whose help was critical in building system software used for this thesis. The Computer Science administrative staff are a wonderful team, I thank Sandy Charette, Bob Ingalls, Terry Hayden, Pam Paslow and Joyce Brock for their support.

Finally, I would like to acknowledge the support and encouragement of my family. My parents have embraced hardship to support our education. My special thanks are due to my elder brother Krishna Kumar, who has supported and encouraged me in all my endeavors. I am deeply indebted to them. My wife, Padma, has stood by me in this endeavor. Her patience and constant encouragement have helped me to complete my thesis. I dedicate this thesis to my loving wife.

ABSTRACT

We propose a new approach to enable synchronous parallel computations adapt to the changing computing environment in networks of nondedicated workstations. Our approach treats the temporary unavailability of a workstation as a transient failure and seeks to reduce its impact on program execution time. It relies on recovering the computations of a failed process by replicating its computations on another available workstation and eventual migration of the recovered process to a new available workstation. Replication of computations is made possible by eagerly replicating the computation state of each process on a backup process. We refer to our approach as the *adaptive replication scheme.*

We analyze the performance of parallel computations using the adaptive replication scheme for exponentially distributed available and nonavailable periods of workstations and outline the conditions under which the scheme is scalable.

The adaptive replication scheme is based on the Bulk-Synchronous Parallel (BSP) model, which is a universal abstraction of a parallel computer. We explore BSP based parallel processing on networks of workstations. Using several applications, we demonstrate that the BSP model can be used to build efficient implementations of parallel algorithms on networks of workstations.

We designed and implemented the adaptive replication scheme on top of a library for the BSP model. We describe adaptive parallel extensions we made to the library to facilitate the design of our scheme. We present a protocol for replication of computation state and recovery of failed processes. To insulate the application programs from the implementation, we designed the adaptive replication scheme in layers. We describe the design of the layers and present their implementation. Finally we present the results of application of the adaptive replication scheme to parallel applications executing on a network of nondedicated workstations.

CHAPTER 1
INTRODUCTION

Recent advances in computers and networking are redefining the way we use computers. Advances in the processing power have brought vast amounts of computing power within the reach of computer users. For a majority of users whose computing needs are small or who need this vast computing power only occasionally, much of this computing power is wasted. At the same time, there exist other users whose computing needs are so large that can only be met with a number of computers working together. Traditionally, solving such large problems required the use of special purpose parallel computers that employed custom designed processors and interconnection networks. However, recent advances in networking technology are allowing the use of computers connected together through local area networks—networks of workstations (NOWs)—to be treated as parallel computers. We can therefore use the computing power of underutilized computers in NOWs to perform useful computations. In this chapter, we identify the trends that are shaping the future of parallel computing and discuss approaches to utilize unused capacity of computers connected through local and wide area networks to perform parallel computations.

Processing power of computers has been growing at a rapid rate. Current workstations[1] have powerful processors that can execute up to several million floating point operations per second. This large computing power is available at a relatively low cost. Additionally, current workstations have large memories allowing for solution of large problems.

Local area networks, which were originally introduced to enable sharing of resources, have grown in size and speed. Computer networks comprising workstations and personal computers, ranging up to several hundreds, have become common in business, research and educational establishments. Thus local area networks (LANs)

[1]We use the term workstation to refer to any computer system designed for a desktop, as done in [3]. Current personal computers are nearly indistinguishable from workstations in their capabilities.

provide us with access to a large number of workstations. Recent advances in networking technology are making fast communication in LANs possible with the use of switched networks that allow bandwidth to scale with the number of processors and low-overhead protocols [3]. Networks of workstations are thus becoming attractive for parallel computations.

With the standardization of programming languages and tools, parallel programming is becoming more general purpose. Message passing libraries such as PVM [69] and MPI [50] allow for portable parallel programs that can run on parallel computers as well as networks of workstations. In recent years, the SPMD (Single Program Multiple Data) paradigm has become the favorite programming paradigm of the parallel processing community. SPMD paradigm allows the programmer flexibility in structuring parallel applications with varying degrees of granularity. This shift toward SPMD programming has enabled networks of workstations to be used as parallel computers in their own right. Current trends in parallel architectures (the IBM SP2, for example) point to use of general purpose workstations connected through dedicated high-bandwidth and low-latency networks. These trends further narrow the gap between parallel computers and networks of workstations. As scientists and engineers tackle larger and larger problems, parallelizing those applications results in large grain parallel computations. A large grain size helps to mask the relatively high latency of networks of workstations.

Even though special purpose parallel computers outperform networks of workstations in parallel efficiency, networks of workstations have certain advantages over special purpose parallel computers. As a result of economies of scale, general purpose computers are relatively inexpensive. Special purpose parallel machines are a niche market and lag the workstations in terms of price-performance. Further, the workstations have access to a large range of software products that need to be ported or customized to run efficiently on the special purpose parallel computers. Perhaps the most important factor is the wide availability of workstations. Almost every programmer has a workstation or a personal computer on her[2] desktop connected to a network. It is desirable to use the vast computing power of networks

[2]The words *his* and *her*, wherever they appear in this thesis, are used to indicate gender-neutral personal pronouns.

of workstations for parallel computing. The NOW project [3] at the University of California at Berkeley aims to demonstrate the use of networks of workstations as the primary computing infrastructure for tasks ranging from interactive to demanding sequential and parallel applications. The project involves building a 100-node network of workstations to demonstrate the capabilities of NOWs for sequential and parallel computing.

The local area networks are connected to form wide area networks to facilitate exchange of data. Millions of computers worldwide are now connected to one another through a hierarchy of interconnections. Thus we now have a large number of machines connected together that can be used for parallel and distributed computations. Even though computers have become more and more reliable over the years, when parallel and distributed applications execute on machines that are not owned by the user, the availability of the computing power for an application is often unpredictable and varying. The unreliability may emanate from link failures or from the usage patterns in force. Whatever may be the source, the unreliability of available computing power affects the performance of parallel programs adversely. In such situations, it is important for the parallel applications to adjust to the dynamically changing computing environment to maintain acceptable performance. We refer to the ability of a parallel application to dynamically adjust to the changing computing environment as *adaptive parallelism*.

1.1 Environments Requiring Adaptive Parallelism

In this section, we consider a few scenarios that require applications to be adaptive to their environments to maintain acceptable performance.

1.1.1 Networks of Nondedicated Workstations

The typical computing facility at a business or a research institution is a network of workstations. The workstations are owned by individual users who use the workstations for a wide variety of tasks ranging from simple editing of files to complex modeling and simulations that require sophisticated graphics and computations. The workstations are powerful enough to handle the occasional compute-intensive tasks, but are underutilized at other times. Further, the computing requirements vary among users; while most users have modest computing requirements, other users have need for computing power beyond the capacity of their own workstations. The computing power from the underutilized workstations can be better utilized to meet the computing demands of other users. Several studies have shown that a large number of workstations in a network are idle at any given time. If this idle computing power can be utilized to perform useful jobs, then considerable computing power will be available to us at low cost. The feasibility of utilizing the idle computing power depends on several factors such as the properties of the communication medium and the statistical characteristics of the use of workstations.

There have been systems that attempt to make use of idle time on workstations. One of the most successful systems for using idle workstations for computations is the Condor distributed processing system [44] developed at the University of Wisconsin. Condor tries to utilize idle workstations by scheduling sequential jobs on them. Piranha [16] is another system for adaptive parallelism and caters to parallel applications with independent tasks that can be scheduled independently of other tasks. One concern in such systems is that the "owner" of a machine should not see any degradation in performance due to the background programs. This concern is usually addressed by requiring that the additional computation be suspended when user activity is detected. The background computation is resumed when user activity ends and the processor is idle. Since these processors are available for use only when they are idle and not available at other times, they are referred to as *transient processors* [37]. We explore adaptive computations on nondedicated networks of workstations in the subsequent chapters.

1.1.2 The Internet

Millions of computers worldwide are now connected to one another through a hierarchy of interconnections. Thus we now have a large number of machines connected together that can be used for parallel and distributed computations. Several researchers are trying to use the Internet for parallel computations [27]. A project to enumerate twin primes using computers at several sites is underway at Rensselaer by Fry et al [28]. Even though currently available wide area networks operate at speeds of a few megabits per second, efforts to develop wide area networks that operate at several gigabits per second are underway. The MAGIC Gigabit Testbed and MAGIC II [17, 34] projects with participating organizations from government, industry and academia are engaged in developing high speed wide area networks. With high speed wide area networks, it is possible to harness this vast computing resource for parallel and distributed computing. However, computers are connected loosely over a wide area network, and the connections are broken frequently thus isolating some of the participating machines from the rest. Parallel and distributed computing over the Internet therefore requires the applications to adapt to the frequent link failures.

1.2 Parallel Computations on Transient Processors

Workstations that are used in a nondedicated manner tend to be either unreliable or available only part of the time and therefore can be characterized as transient processors. Parallel computation of a task requires that the task be divided into subtasks, which are then executed on a set of processors in parallel. We refer to the subtasks as *components* and the processes that execute the subtask as *component processes* of the parallel computation. Each component of the parallel application is assigned to a processor. A component process is scheduled when its host processor is idle and is suspended when the host processor is busy.

1.2.1 Impact of Transient Processors on Program Execution Time

Kleinrock and Korfhage [37] analyzed the impact of transient processors on program execution time. They analyzed the probability density of a program's finishing time on both single and multiple processors for random values of *available* and *nonavailable* periods from a general distribution. They model a program as consisting of a number of stages, each of which must be completed before the start of the next. Each stage represents a deterministic amount of work. They make some simplifying assumptions. Their analysis ignores overhead that occurs in a real system such as communication and processing delays and thus their analysis yields an optimistic estimate of the program performance. They obtain expressions for the finishing time density of the program under the assumption that the duration of a stage is large compared to the mean lengths of available and nonavailable periods. They try to lessen the impact of variance in processor available and nonavailable periods by smoothing (i.e., by mapping large units of computation). Smoothing works well with exponential distribution, in which the frequency of long periods quickly tends to zero. Assuming smoothing for a general distribution, they found that the mean value of finishing time was proportional to the inverse of the fraction of processor idle time. They note that when the duration of a stage is not large compared to the mean lengths of the available and nonavailable periods, the mean value of the finishing time remains the same but the variance increases.

Recent research on statistical properties of host load in distributed environments by Dinda et al [24] indicates that host load can vary drastically and the ratio of maximum to mean values can be very high. Their traces of host load exhibit a high degree of self-similarity indicating that load varies in complex ways in all time scales. Their measurements demonstrate that the assumption of exponential distribution of host available and nonavailable periods is not valid in many environments. Based on their measurements, they conclude that smoothing may not be very effective since variance may not decline with increasing smoothing intervals as quickly as expected. In view of their observations, for the results obtained by Kleinrock et al for the finishing time of a program on transient processors to be valid, the duration of the program stages need to be extremely large compared to the maximum lengths of available and nonavailable periods. This requirement can not be satisfied for many practical problems because of two reasons. First, large problems require several hundreds or thousands of synchronized computation steps to arrive at the solution. As a result, each computation step has to be short in order for the overall computation to finish in a reasonable time. For example, scientific computations like plasma simulation require a large number of iterations to reach steady state, with each iteration followed by synchronization necessitated by sharing of computation results by the processors. Numerical computations, likewise, require thousands of iterations with associated synchronization to converge to a solution. Second reason is that the size of the problem that can be solved is often limited by the memory available to the computation on each processor and this limits the amount of computation in any stage. In chapter 7, we discuss a graph search problem that is limited by the memory available on each processor for the adjacency matrix partition. Due to the short duration computation steps in the applications we consider, smoothing to reduce the variance in the finishing time of the application is not an option. Instead our approach is based on migrating tasks from unavailable to available machines. In a synchronous computation, execution time of a computation step is the maximum of the execution times of the step on the participating processors. Frequently synchronizing parallel applications, due to the short duration of each computation step and the large number of computation steps in the application, are severely impacted by the nonavailability of one or more participating host machines. Our goal is to reduce overall finishing time of frequently synchronizing parallel applications executing on transient processors using a scalable adaptive parallelism scheme.

1.3 The Bulk-Synchronous Parallel Model

The Bulk-Synchronous Parallel (BSP) model [70], introduced by Leslie Valiant, is a universal abstraction of parallel machines which can be used to design portable parallel software. The model was expected to fulfill two purposes: the model should lend itself to theoretical analysis and it should be possible to implement the model efficiently. The model is expected to fill the gap between theory and practice, thereby enabling the development of general purpose, architecture independent parallel programs. The model offers a framework within which we can unify the various classes of parallel computers — distributed memory architectures, shared memory multiprocessors and networks of workstations.

The model has been attracting attention from a number of researchers. Bisseling and McColl [9] investigate the efficiency of several computations in linear algebra that include iterative solution of sparse linear systems, partial differential equations, etc. Their analysis suggests that knowledge about the underlying structure of the problem is important to achieving efficient parallel computations on a

BSP computer.

Several people are working on providing suitable environments for BSP programming. Miller developed the Oxford BSP library [48, 49], a simple yet robust library for BSP programming. The library uses an SPMD model with static allocation of processors. BSPlib [25] is another programming library based on the BSP model developed by the Oxford Parallel group. BSPlib is a part of the Oxford BSP toolset which includes profiling tools and implementations of BSPlib for several machine architectures. Like the Oxford BSP library, the BSPlib supports SPMD parallelism and is based on efficient one-sided communications. H-BSP [18] is a general purpose parallel computing system, currently under development at Harvard university. The system consists of (a) a high level programming language BSP-L, (b) a collection of compiler tools (optimizers, code generators, etc.) which generate efficient code for a large range of parallel computers based on the parameters of the model, and (c) a collection of library operations for communication and synchronization for a runtime system.

By expressing the cost of an algorithm in terms of a few parameters, The BSP model allows us to determine the cost of an algorithm and predict its performance on any parallel architecture for which the values of the parameters are known. Thus the BSP model enables predictable parallel computing. A more detailed description of the model is given in Section 3.1.

1.4 Research Objectives and Approach

We are now ready to specify the objectives of our research. Stated broadly, our primary objective is to investigate techniques and algorithms that enable parallel applications to adapt to the changing computing environment to maintain performance and to make use of the available computing resources efficiently. The techniques to make the parallel applications adaptive should be general purpose and independent of any machine architecture. Further, we require that the techniques and algorithms we develop be scalable.

Towards this end, our approach to parallel processing is based on the Bulk-Synchronous Parallel model described in the previous section. Thus an investigation of BSP-based parallel processing forms the first step in our research. Our approach to adaptive parallel computations is described in Section 1.4.2.

Since different computing environments pose different challenges, a successful approach to solving the problem requires exploring strategies that fit each specific environment. With this objective in mind, we wish to investigate the specific problem of utilizing idle time on workstations for parallel computations. As seen in the previous section, there have been some efforts to make use of idle computing power of a network of workstations. However, all systems known to the author, with the exception of Stardust [12], focus on either sequential jobs (Condor [44]) or parallel applications with independent tasks that synchronize infrequently (Piranha [16]). Stardust uses an approach similar to ours. Our system differs from Stardust in how quickly the additional computation is suspended upon detecting user activity. These systems will be discussed in more detail in chapter 2. In our research, we aim to explore the most challenging case of frequently synchronizing parallel applications on transient processors.

In general, we subscribe to the notion that a user of a workstation should not be inconvenienced by the execution of additional background jobs on his/her workstation. Hence, in our approach, we require that any background parallel computations be suspended or preempted from a workstation when the user returns. On the other hand, we are open to using a few cycles to improve the adaptive behavior of our algorithms, if it can be done without putting too much load on the user's machine. We can now describe our approach to adaptive parallel computations on networks of workstations.

1.4.1 Our Approach to Adaptive Parallel Computations on NOWs

Towards the goal of reducing the impact of the unavailability of a processor on the execution time of the application, we treat the unavailability of a processor as a failure. Specifically, a transition of the host processor from an available to a nonavailable state will be referred to as a *transient failure* of the component process. The effect of a transient failure is to delay the parallel application. Conversely, a transition of the host processor from a nonavailable to an available state will be treated as *recovery* of the suspended component process. In the following discussion, we assume that transient failures of processors are independent events.

In general, there are two ways to deal with failures in any system: prevention (or avoidance) and recovery. Based on this general principle, we identify three strategies to deal with transient failures. An analysis of these strategies is the first part of our investigation.

Our first approach to reduce the impact of transient processor failures is based on redundancy. Given the probability of failure (nonavailability) of a processor, the probability of two processors failing together is much smaller. Therefore, we can reduce the probability of failure of a component by replicating the component on more than one processor. We refer to the processes as *replicated processes.* Our justification to considering the idea of replicating components is based on our assumption that the idle time on a processor is free for use and therefore, costs nothing.

The second approach is based on the concept of *recovery from failures.* Specifically, we would like to recover the computations of the failed process in another component process on a different processor. We further assume that at least one of the component processes is immune to the transient processor failures. This assumption is easily satisfied, since it is possible to place at least one component of the parallel computation on a workstation owned by the user. We refer to this process as the *master process.* The computations of the failed component process can be recovered by sending the computation state of the failed process to the master process. The master process can use this data to recreate the computation state of the failed process and execute its computations. This approach requires the services of the master process for each process that failed. Consequently, the master process can become a bottleneck in case of multiple transient failures.

Our third approach tries to deal with transient failures preventively. This is done by communicating the state of each component process to a neighbor at the beginning of the computation step. Each component process will therefore be capable of recovering the computations of the sender process in the event of failure of the sender process. Recovery of computations in this approach is distributed among the components.

Scheme	Replication of Computations	Replication of Data	Migration of Processes
Straightforward execution	No	No	No
Full process replication	Eager	Lazy	Not needed
Standard failure recovery	Lazy	Lazy	Needed
Adaptive replication	Lazy	Eager	Needed

Table 1.1: Classification of schemes for transient failure recovery using replication of data and computations and migration of processes

These approaches try to mask or reduce the impact of processor state transitions by replicating processes, computations and/or data to varying degrees. They can be classified based on the eagerness with which they replicate processes, computations and/or data. Table 1.1 shows the different approaches.

The straightforward execution simply delays the completion of the computational step until all participating processes finish their computation, even if some of the processors change their state from available to nonavailable. The *full replication scheme* eagerly replicates computations by actively replicating each component process on more than one processor, thereby increasing the chances of at least one of the replicas finishing the computation successfully. This scheme uses (lazy) data replication to enable replicas that have fallen behind to catch up with the leading process that has finished its computation. The third scheme is based on a *failure recovery* strategy. There is no replication of processes. In the event of failure, the failing process sends its computation state to a reliable process, which replicates the computations of the failed process after reconstructing its state. In the *adaptive replication scheme*, the computation state is eagerly replicated on a neighbor process at the beginning of a computation step. In the event of a failure of the sender process, the receiver process uses the state data it received to replicate the computations of the failed process. We analyze these approaches in more detail in the following sections, and compare them by analyzing the finishing time of a parallel program under each approach.

1.4.2 Adaptive Parallel Computations in the BSP Model

Our view of parallel computation is based on the Bulk-Synchronous Parallel Model [70]. A brief introduction to the BSP model is given in section 1.3. A more detailed description of the model is given in 3.1. Computation in the BSP model is a sequence of parallel supersteps consisting of computation and communication operations. All the processors participating in the parallel computation synchronize at the end of each superstep. The BSP model has been used to implement a wide variety of scientific applications including numerical algorithms [9], combinatorial algorithms [30] and several other applications [13]. We used the model to build an efficient implementation of plasma simulation on a network of workstations [54] as described in chapter 3. By basing our approach on the BSP model, we are able to analyze the adaptive replication scheme and describe its performance. Chapter 4 presents an analysis of the adaptive replication scheme in terms of network and program parameters.

Despite these benefits, there still exist concerns about the efficiency of BSP implementations. The barrier-synchronization at the end of a BSP superstep can be expensive. However, by exploiting knowledge about the communication patterns in an application and by overlapping communication with local computation, the cost of global synchronization can often be reduced. Barriers also have desirable features for distributed system design. By making circularities in data dependencies impossible, they avoid potential deadlocks and live locks thus eliminating costly detection and recovery. Barriers ensure that all processes reach a globally consistent state which allows for novel forms of fault tolerance [65]. In our model of parallel computation based on BSP, the participating processors are all in a globally consistent state at the beginning of each computation step which eliminates the need for consistent checkpointing. The synchronization at the end of a superstep also provides a convenient point for checking process[3] failures. Should one or more processes fail, the surviving processes can start the recovery of the failed processes at this point.

For these reasons we base our approach to adaptive parallelism on the BSP model. Our approach employs a combination of replication of computation state and

[3]Throughout the rest of the paper, we use the term *process* to refer to a component process of the parallel computation.

replication of computations. We refer to it as the *adaptive replication scheme (ARS)*. The following chapters describe our approach to adaptive parallelism including our assumptions, the protocol for replication of computation state and recovery of failed processes, the role of process migration, and the algorithm that implements the adaptive replication scheme.

1.5 Summary

In this chapter we introduced current trends in parallel computing and discussed their impact on parallel computations on networks of workstations. We proposed adaptive parallelism as a means for parallel computations to deliver acceptable performance in the face of a changing computing environment. We described two scenarios which require parallel applications to adapt to their computing environments. These environments can be described as a network of transient processors. We discussed the impact of transient processors on frequently synchronizing parallel computations. We introduced the Bulk-Synchronous Parallel model and described our approach to adaptive parallel computations within the framework of the BSP model.

1.6 Thesis Outline

In subsequent chapters we expand on the topics and issues briefly discussed here.

In chapter 2 we discuss related work in this area including workstation usage patterns, remote execution facilities, remote execution policies and scheduling and some recent work in adaptive parallel computing on networks of workstations.

In chapter 3 we discuss BSP-based parallel computing on a network of dedicated workstations. We discuss the characteristics of a network of workstations when viewed as a BSP computer. With the example of an application for plasma simulation, we demonstrate the use of the BSP model in analyzing and then tuning the performance of a parallel application for a given parallel architecture. We also briefly discuss two other parallel applications: a graph search problem and an application of finite element method in the modeling of bioartificial artery.

In chapter 4 we consider parallel computations running on a network of transient processors. We characterize nondedicated workstations as transient processors and consider approaches to alleviate the impact of transient processors on program execution time. We then analyze the performance of our adaptive parallelism approach and investigate the conditions under which the approach is scalable.

In chapter 5 we consider adaptive parallelism in the BSP model and describe the protocol for replication of data and recovery of failed computations. Chapter 6 describes the extensions to the Oxford BSP library, the design of the adaptive replication scheme in layers and an implementation of a prototypical system.

In chapter 7 we present the application of the adaptive replication scheme to two parallel applications and present performance results. We also present a demonstration of the scheme in a real environment.

In chapter 8, we discuss the work in a broader perspective including its scope and limitations, summarize the results and our contributions and discuss possible directions for future work.

CHAPTER 2
RELATED WORK

Much research has gone into exploring ways of effectively utilizing the computing capacity in networks of workstations. The work done is mainly in the following areas: analysis of workstation usage patterns, policies and scheduling algorithms to allocate remote execution capacity, development of remote execution facilities, message passing libraries to aid parallel computations on NOWs and analysis of the impact of transient processors on program execution time. We summarize related work in each of these areas. We also describe the Bulk-Synchronous Parallel model which aims to unify various parallel architectures including networks of workstations into one model.

2.1 Workstation Usage Patterns

Mutka et al analyzed the workstation usage patterns and their availability for remote execution [52]. They monitored a network of workstations consisting of 13 workstations over a period of 5 months. The workstations observed were owned by a variety of users: faculty, systems programmers and graduate students. They obtained the profile of *available* and *nonavailable* periods of workstations. A workstation is considered unavailable if the average cpu usage was above a threshold value (one-fourth of one percent) within the last 5 minutes. A workstation was considered available at all other times. An analysis of the traces showed that the workstations were available approximately 70% of the time. The average available and nonavailable periods were of duration 100 minutes and 40 minutes respectively. Long available periods are good for scheduling background jobs on the machines. The most surprising result was that the long available periods were observed even during the busiest times. The average amount of time the workstations spend in available state during the busy times was 50%. Their results show that workstations are good candidates for remote execution of jobs.

Krueger and Chawla [38] made observations over a 3 month period of a network consisting of 199 Sun-3/50 diskless workstations. Some of the machines were privately owned, while others were located in public laboratories. They found that, on the average, 91% of the computing capacity of the network of workstations is idle. The overall range of observations was from 77% to 96% which shows that a significant variation in unused capacity is possible. Even during the periods that the network was most heavily used, unused capacity rarely fell below 87%. However, even during the periods of low overall utilization, some workstations were heavily used while many others were idle. As a result, significant delays can occur at heavily used workstations at times when other workstations could provide immediate service. Therefore, significant improvements in response times can be obtained by transferring some of the jobs from the heavily loaded machines to the idle machines.

Dinda and O'Hallaron [24] conducted research into statistical properties of host load in distributed environments. They collected traces of load on a number of machines over a period of one week. Their data indicates that host load can vary drastically. In addition, host load traces exhibit a high degree of self-similarity and as a result, lessening the effect of load by smoothing (not migrating long-running tasks, for example) may not be very effective.

Mutka and Livny [52] made actual measurements of a network of transient processors. They developed models for the available and nonavailable period densities to fit these measurements. Their model for the distribution of available periods has a mean of 91 minutes and a variance of 40225 minute^2. The probability distribution function of nonavailable periods has a mean of 31 minutes and a variance of 2132 minute^2. These mean values were also used by Kleinrock et al [37] in examples to illustrate results of their analyses. In our experiments using simulated transient processors (section 7.3), we use values in the range reported by Mutka et al for mean lengths of available and nonavailable periods.

2.2 Remote Execution Facilities

There has been much research on systems that make use of idle time on workstations through load balancing and process migration. The systems described in this section cater to sequential jobs and independent tasks and not to parallel computations with synchronization. We summarize some of the systems here.

Condor ([44, 45, 10, 42]) is a very successful remote program execution facility developed at the University of Wisconsin. The system schedules long running background jobs on idle workstations. When the owner of a workstation resumes activity at a workstation, Condor checkpoints the remote job running on that workstation and later migrates it to another workstation when one is available. The system guarantees that the job will eventually complete. Condor tries to improve the utilization of the network by accommodating the requests from heavy users for extra computing power without curtailing the freedom of light users. The system has been operational at the University of Wisconsin and at many other sites worldwide.

Condor uses the Remote Unix facility [43] under which a *shadow* process runs locally as a surrogate of the process running on the remote machine. Any Unix[1] system call made by the remote program is communicated to the shadow process. The Remote Unix facility is responsible for checkpointing the remote program.

The Sprite network operating system [61] uses a "workstation" model, in which a user executes tasks primarily on his/her own machine (called the "home machine"). It provides a mechanism to take advantage of idle machines transparently using migration. Logically, a process in Sprite executes on the user's home machine. However, the process can physically migrate to another processor at any time. Transparency is assured by forwarding location-dependent operations to and from a process's home machine.

Amoeba [51] is a distributed operating system. Its system architecture is organized around a "processor pool". All users have equal access to all pool processors. Pool processors are dynamically allocated to processes as needed. Amoeba achieves some load balancing by assigning the most desirable processor to a process.

The V System [19] uses a workstation model similar to Sprite. It uses process migration to execute new tasks on lightly loaded workstations.

There are many other systems that attempt to make use of idle computing power through a remote execution facility. The Benevolent Bandit Laboratory (BBL) [26] runs distributed computations on a network of personal computers. A

[1]Unix is a trademark of AT&T Bell Laboratories.

special shell runs on each machine. When the machine is at the operating system prompt level, it is available for use. If a user starts using a machine that is a part of background distributed computation, a replacement machine is chosen from the pool of the idle processors. Lyle and Lu [46] describe a simple remote program execution facility that runs at shell level. The Butler [58] system at Carnegie-Mellon University also provides remote execution facilities through servers that run on idle workstations instead of on a fixed workstation. In the Butler system, the state of the job is not preserved when the job is preempted. As a result, all the work accomplished by a job could be lost. The Worm program [64] at Xerox PARC was one of the early experiments in distributed computing using idle workstations.

2.3 Remote Execution Policies and Scheduling

In a system where idle machines are allocated to long running jobs, users who request additional computing capacity occasionally must be protected from users who try to acquire all the additional capacity available. Mutka and Livny [53] proposed the *Up-Down* algorithm designed to allow fair access to remote capacity for those who use the system lightly in spite of large demands by heavy users. This is achieved by maintaining a *schedule index* for each workstation. The index is increased when remote capacity is allocated to the workstation. When a workstation wants remote capacity but is denied access to it, the index is decreased. The priority to remote capacity of a workstation is determined by its schedule index. When a station has a higher priority job to execute, and if there are no idle stations, the coordinator preempts a remotely executing job from a station with lower priority.

The scheduling strategy used in Condor [44] is a mixture of centralized and distributed approaches. Each workstation has a local scheduler and a background job queue. The jobs submitted by the user at this workstation are placed in the job queue. In addition, there is a central scheduler at one of the workstations. The central scheduler polls the workstations periodically and gathers information on which workstations have idle capacity for remote execution and which workstations have background jobs waiting. The central coordinator allocates capacity from idle workstations to local schedulers on workstations that have background jobs waiting.

The local schedulers are responsible for scheduling their own jobs. If there is more than one job in the job queue, the local scheduler decides which job should be next. Between successive polls by the central coordinator, each local scheduler monitors its station to see if it can serve as a source for remote capacity. When a background job is running, the local scheduler periodically checks for user activity. As soon as user activity is detected, the local scheduler preempts the background job immediately.

The Sprite network operating system accords priority to the owners of the workstations. When a owner returns, any *foreign* processes on this workstation are migrated to other workstations. A workstation is also reclaimed when there are no more idle workstations and one process is using more than its fair share of workstations. In Sprite, an application may use another workstation only if the workstation is idle and no other application is already using it.

The criterion used by many systems (including Condor) treats availability of workstations for remote execution in an "all or nothing" fashion - if a node is idle then it is a candidate for executing a remote job, otherwise it is not. In this approach, any machine can take over an idle machine for execution of its jobs, but as soon as the owner of the machine uses it, the remote jobs have to be suspended. The remote jobs are either put in the background, moved to another idle machine [44] or just killed [58]. While these techniques guarantee the ownership of the resources of an idle machine to the owner of the workstation, they do not assure any performance improvements for the remote jobs the idle node may be servicing.

Alonso and Cova [2] replace this approach with a gradual one, called "High-Low." In this approach, each machine in the network determines the amount of sharing it is willing to do. The node will share some of its resources as long as its users are not significantly affected. The policy works as follows: each time the execution of a job is requested at a node, the load of the machine is compared against its High-mark value. If the former is higher than the latter, then the load sharing mechanism will try to execute the job at a remote node. On the other hand, whenever a request to execute a remote job arrives at a machine, the request is accepted only if the load on the machine is less than its Low-mark value. When the High-mark value is greater than the Low-mark value, these thresholds divide the range of possible load values into three regions: overloaded (above the High-mark), normal (above the Low-mark and below the High-mark), and under loaded(below the Low-mark). When the load of a machine is in the overloaded region, new local jobs are sent to be run remotely and remote execution requests are rejected. In the normal region, new local jobs are run locally and remote execution requests are rejected. In the under loaded region, new local jobs are run locally and remote execution requests are accepted. Their system does not implement migration of the jobs once they are started on a machine.

The V System executes remote tasks at a lower priority than local ones in order to reduce their impact on interactive response.

2.4 Adaptive Parallel Computations on NOWs

While all the systems mentioned in the previous subsections are successful to varying degrees in utilizing the idle computing power, they all address the placement of independent jobs on idle machines. Specifically, they do not address the issue of running tightly coupled parallel programs whose components synchronize periodically. This is the issue of using a network of workstations as a parallel computer. There have been several message passing libraries for distributed memory computers, which can be used to program networks of workstations.

PVM [69] (Parallel Virtual Machine) is a software package that allows a heterogeneous network of parallel and serial computers to appear as a single concurrent machine. The PVM user library [29] provides routines for initiating processes on other machines, for communicating between processes, and changing the configuration of machines.

Otto et al [35] discuss three systems that transparently migrate load among the workstations in the network for PVM applications. Migratable PVM (MPVM) uses Unix processes as its virtual processors just like PVM, and allows transparent migration of these processes. The processes of a PVM application can be migrated without any help from the application program. This system uses a global scheduler to coordinate migration events. UPVM is a virtual processor package that supports multi-threading. The virtual processors are called User Level Processes (ULPs).

ULPs are light-weight UNIX-like processes and can be migrated. Adaptive Data Movement (ADM) provides programmers with constructs for developing adaptive computations based on work re-distribution. Work redistribution is achieved by data movement. To maintain consistency of the parallel computation, MPVM and UPVM flush all messages to and from the process to be migrated. Then the computation state of the process is transferred to a new skeleton process that is started on the target machine. Our approach differs from the PVM based systems in the following ways. By basing our approach on the BSP model, we avoid the need for a consistent checkpoint. Thus our approach is simpler than the PVM based approaches. In our approach, recovery of a failed process is performed on another available machine and eventually migrated from this machine to the target machine. Thus migration of the process imposes no overhead on the unavailable machine.

The message passing interface standard MPI [50, 11, 66, 33] was proposed by MPI Forum, a consortium of industry and research institutes. Its goal is to serve as a standard for writing message-passing programs. The interface is suitable for SPMD style of programming. MPI has been implemented on a variety of platforms including networks of workstations.

Parform is a high performance platform for parallel computing based on a network of workstations, developed at the University of Zurich by Cap and Strumpen [14]. It involves three kinds of processes intended for programming the parallel computations: *administrative process*, *executive processes* and *load sensor processes*. The programmer supplies the application dependent portion of administrative procedure and the executive procedure. The platform specific part of the administrative process is supplied as part of a library. This portion, together with the load sensors, is responsible for collecting load data and identifying idle machines. The application specific portion of the administrative process divides the computation among the executive processes and collects results. The application independent code of the executive process sets up the communication links and supports message passing and dynamic load balancing. The application dependent part of the code performs the arbitrary size subtasks.

In [15], Cap and Strumpen address the issue of utilizing heterogeneous networks of workstations with constantly changing load situation for parallel computations. In frequently synchronizing computations, the throughput of a heterogeneous network is reduced to that of the slowest workstation. They solve this problem using *heterogeneous partitioning* and *dynamic load balancing.* In heterogeneous partitioning, the task is divided according to the performance of the individual workstations. As a result, workstations able to contribute more to a parallel computation are automatically assigned larger subtasks, leading to better performance from the heterogeneous network. However, due to the dynamically changing load situation, the available performance from the individual workstations will change eventually. They solve this problem using dynamic load balancing. At certain times during the computation, processes exchange the load values of their hosts and adapt the size of their subtasks to the actual load situation. This is achieved by communicating parts of the subtasks from processes on hosts with increased load to processes on hosts with less load. However, before actually changing the size of subtasks, a protocol must ensure that the communication partner is ready to receive additional chunks of work with respect to its own resource and load situation. This approach requires mechanisms to change the size of a subtask during the computation depending on the load situation of individual machines.

Carriero et al [16] implemented an adaptive parallel system called *Piranha*, which works with Scientific Computing Associates' tuple-space based coordination model *Linda.* A tuple space is a virtual shared, associative, object memory accessible to all nodes within a parallel computing environment. Tuple space supports uncoupled communication between producers and consumers of tuple data; producers and consumers of tuple data (which are the subtasks of an adaptive parallel program) need not be aware of the identity of each other. The subtasks can therefore freely move around the network in search of idle machines.

Piranha implements an adaptive version of master-worker parallelism. The number of workers can vary as the computation proceeds. The master process is persistent and runs on the node from where the application is started. It creates the tasks that need to be executed. Whenever a node becomes idle, the system creates a new worker process on the node. Each worker process executes a loop, in which it claims a task from the tuple space, executes it and adds the result to the tuple space. When the owner of the node returns and the node is busy, the worker process retreats. The Piranha runtime system consists of daemons, one per participating node. The overhead due to the daemons is insignificant.

In Piranha, the user has to supply the routines for the master process, worker process and the retreat function. Tasks are represented by task descriptors. Inter-task dependencies are handled by the dependency information stored in the tuple space. A number of applications have been bench marked on Piranha. The observed efficiency, measured as a ratio of sequential time to parallel time, is quite high (above 90%). However the applications bench marked are all coarse grained and only the process time spent on the problem were considered in the measurements. Master-worker parallelism is suitable only when the tasks of the parallel computation are independent. This approach therefore does not apply to synchronous parallel computations.

Stardust [12] is a system for parallel computations on a network of heterogeneous workstations. It captures the state of the computation at the barrier synchronization points in the parallel program. In saving the application state, only the architecture-independent data is saved on to disk and transferred to other nodes which allows for migration of the application between heterogeneous hosts. A major limitation of Stardust's mechanism of using naturally occurring synchronization barriers is that it limits the number of points where an application can be stopped and migrated.

2.5 Summary

In this chapter we reviewed previous work relating to utilizing idle workstations for sequential and parallel computations. The systems for adaptive parallelism fall into three categories. Systems in the first category try to dynamically balance the load among the participating processors by redistributing the work through data movement. ADM for PVM applications and Parform belong to this category. The second category consists of systems that cater to parallel computations with independent tasks. Since the tasks are independent, they can be restarted on new machines. Piranha belongs to this category. In the third category, systems try to dynamically map a set of virtual processors to the currently available processors. When a workstation becomes unavailable, the virtual processor on this workstation must be migrated to a new available machine. These systems differ in the way they try to maintain the consistency of the parallel computation and in the way they migrate processes from one machine to another. MPVM and UPVM flush all messages to and from the process to be migrated. Stardust uses naturally occurring barriers to migrate processes without losing consistency. Our approach falls into this category, but differs from the others in the following respects. Our approach is based on the abstract parallel computer model of BSP. The synchronous nature of BSP eliminates the need for maintaining the consistency of the parallel computations. This makes our approach much simpler compared to the other approaches. Unlike Stardust, our scheme does not limit the number of points where a process can be suspended. In addition, our approach differs from all other approaches in that recovery of a failed process (a process on a workstation that has become unavailable) and eventual migration to a new available workstation are performed on an available machine and not the currently unavailable machine. Thus recovery of the failed process and migration to an available host impose no additional overhead on the host that has become unavailable. Thus our approach is less intrusive compared to other approaches. Finally, basing our model on BSP allows us to analyze the performance of a parallel computation using our approach in terms of the program and network characteristics.

CHAPTER 3
PARALLEL COMPUTING ON NOWS USING THE BULK-SYNCHRONOUS PARALLEL MODEL

The efficiency of parallel computations depends on the properties of the communication medium such as latency and bandwidth. In general, local area networks connecting the workstations tend to have higher latency and lower bandwidth compared to the interconnection networks in special purpose parallel computers. The higher latency and lower bandwidth tend to reduce the efficiency of parallel computations. Parallel computations intended to run on workstations require careful design to maintain parallel efficiency.

The Bulk-Synchronous Parallel model [70] introduced by Leslie Valiant is a universal abstraction of a parallel computer. The model is intended to support creation of portable parallel software by acting as a bridging model for parallel computation — a model between hardware and a programming model to insulate software and hardware development from one another.

In this chapter, we characterize the network of workstations as a BSP computer and describe an implementation of plasma simulation for networks of workstations using the BSP model. We illustrate, with the example of plasma simulation, how the BSP model can be used to analyze and gain insight into the structure of the parallel computation, and use that knowledge to design an efficient implementation of plasma simulation for networks of workstations.

3.1 Bulk-Synchronous Parallel Model

The Bulk-Synchronous Parallel model [70] consists of the following attributes:

- *components* (processors) which execute programs
- a *router* that provides point to point communication between pairs of components, and

- a *synchronization mechanism* to synchronize all or a subset of the components at regular intervals. The *periodicity* parameter L represents the minimum time between synchronizations.

A computation consists of a sequence of *supersteps*. In each superstep, a component performs some local computation and/or communication with other components. The data communicated are not guaranteed to be available at the destination until the end of the superstep in which the communication was initiated.

In analyzing the performance of a BSP computer, a *time step* is defined to be the time required for a component to perform an operation on data available in the local memory. The performance of a BSP computer is characterized by the following set of parameters:

- number of processors (p),
- processor speed (s),
- synchronization periodicity (L), and
- a parameter to indicate the global computation to communication balance (g).

s is the processor speed in number of time steps per second. L is the minimal number of time steps between successive synchronization operations. g is the ratio of the total number of local operations performed by all processors in one second to the total number of words delivered by the communication network in one second. It should be noted that the parameters L and g usually are not constants, but functions of the number of processors p. These functions are in turn defined by the network architecture and the implementation of the communication and synchronization primitives.

For the analyses presented in this and other chapters, we assume a network of workstations with a fixed bandwidth communication medium such as an Ethernet. Communicating large amounts of data on networks of workstations using a fixed bandwidth communication medium will cause the interconnection network to sequentialize message flow from/to the active processors. Under this assumption, the parameter g can be expressed as a function of the number of processors as follows:

$g(p) = g_0 p$, where g_0 is a constant. We also assume a synchronization mechanism that uses a tree structure, and therefore we have the following relationship for the parameter L. $L = L_0 \log(p)$, with L_0 being a constant.

BSP parameters allow the user to analyze the complexity of a BSP algorithm in a simple and convenient way. The complexity of a superstep, S in a BSP algorithm is determined as follows. Let w be the maximum number of local computation steps executed by any processor during the superstep. Let h_s be the maximum number of messages sent by any processor and let h_r be the maximum number of messages received by any processor during the superstep. In the original BSP model, the cost of S is given by $\max\{l, w, gh_s, gh_r\}$ time steps. An alternative formula for the complexity of a superstep [31] is to charge $\max\{l, w + gh_s, w + gh_r\}$ time steps for the superstep. Yet another definition [9] charges $l + w + g\max\{g_s, g_r\}$. Different cost definitions reflect different assumptions about the implementation of the supersteps, in particular about which operations can be done in parallel and which ones must be done in sequence. The last formula assumes that the local computation, communication and synchronization are done in sequence. The difference is not crucial, since the cost of a BSP algorithm computed as the sum of the costs of the supersteps using either of the above formulae differ only in the constant.

By designing algorithms that are characterized by the size of the problem (n), the number of processors (p) and the two parameters that characterize the performance of the communication network (l and g), we can ensure that the algorithms can be efficiently implemented on a range of BSP architectures. Such a design leads to architecture independent BSP algorithms [9].

3.1.1 Network of Workstations as a BSP Computer

In terms of the BSP parameters, parallel computers are often characterized by large values of s (fast processors) and low values of L and g (a communication network with low latency and large bandwidth). A general purpose network of workstations, on the other hand, is characterized by values of s that are somewhat lower and values of L and g that are much larger than the corresponding values for the parallel machines (high latency and low bandwidth due to the loosely coupled nature of these networks). Networks of workstations, with relatively high values of l and g, often require algorithms to be designed differently from algorithms that are designed to run efficiently on parallel computers. In designing algorithms for a BSP computer with a high value of g, we must ensure that for every non-local memory access, we perform approximately g operations per local data.

As an example, consider the task of broadcasting data from a single processor to all other processors using the point to point communication primitives. In a parallel computer, broadcasting of data is often performed by using a (binary) tree structure with the processor initiating the broadcast at the root of the tree and the other processors occupying the other nodes. In the first superstep, the processor at the root node communicates the data to the processors at its child nodes in the first level. In each subsequent step, processors at nodes in the currently active level communicate the data to processors at their child nodes in the next higher level. The communication is increasingly parallel as data move from the root of this tree to the leaves. We refer to this scheme as *logarithmic broadcast*. The communication in the opposite direction (from the leaves to the root) implements data gathering. Both operations take number of steps proportional to the logarithm of the number of processors involved.

The cost of logarithmic broadcast of h units of data on NOWs is

$$L\log(p) + g(p-1)h = L_0\log^2(p) + g_0(p-1)h \tag{3.1}$$

In the *linear broadcast*, the broadcasting node simply communicates the data to all other nodes in a single superstep. Hence, the cost of the linear broadcast of h units of data is

$$L + g(p-1)h = L_0\log(p) + g_0(p-1)h \tag{3.2}$$

Comparing 3.1 and 3.2 shows that, unlike in a parallel computer environment, linear broadcast is always faster in a NOW environment. However, when logarithmic gather is used, computations can be performed on the data being broadcast in parallel at the nodes of the tree, whereas linear gather forces computations to be delayed until all of the data arrives at a processor. This feature may make logarithmic gather more attractive than linear gather under some circumstances. For example, using logarithmic gather, summation can be performed on the data being gathered.

3.1.2 The Oxford BSP Library

The Oxford BSP Library [48, 49], developed by Richard Miller, is used to implement the plasma simulation on NOWs. It is based on a slightly simplified version of the model presented in [70]. The programming model uses a SPMD program style, and static allocation of processors. The most significant feature of the library is its use of remote assignment semantics for non-local data access, which simplifies debugging. The library is small, simple to use, yet robust. Figure 3.1 lists the library functions for C programs.

The Oxford BSP Library

- Process Management
  ```
  void bspstart(int argc, char* argv[], int maxprocs, int* nprocs, int* mypid);
  void bspfinish();
  ```
- Synchronization Management
  ```
  void bspsstep(int sstep_id);
  void bspsstep_end(int sstep_id);
  ```
- Communication Management
  ```
  void bspfetch(int pid, void* src, void* dst, int nbytes);
  void bspstore(int pid, void* src, void* dst, int nbytes);
  ```

Figure 3.1: The Oxford BSP Library C Functions.

3.2 Plasma Simulation on NOWs Using the BSP Model

Norton et al [60] implemented a plasma simulation on a distributed memory parallel machine. We discuss the implementation of the simulation on a network of workstations using BSP after giving an overview of the particle in cell (PIC) method used in the simulation.

3.2.1 Overview of Plasma PIC Simulation

A material subjected to extreme heat, pressure or electric discharges undergoes ionization, where the electrons are stripped from the atoms, acquiring free motion. The created mixture of heavy positively charged ions and fast electrons forms an ionized gas called a plasma. Familiar examples of plasma include the Aurora Borealis, neon signs, the ionosphere, and solar winds. Fusion energy is also an important application area of plasma physics research. The plasma Particle In Cell simulation model [8] integrates in time the trajectories of millions of charged particles in their self-induced electro-magnetic fields. Particles can be located anywhere in the spatial domain; however, the field quantities are calculated on a fixed grid. Following [60], our simulation models only the electrostatic (coulomb) interactions.

The General Concurrent Particle in Cell (GCPIC) Algorithm [40] partitions the particles and grid points among the processors of the MIMD (multiple-instruction, multiple-data) distributed-memory machine. The particles are evenly distributed among processors in the primary decomposition, which makes advancing particle positions in space and computing their velocities efficient. As particles move among partitioned regions, they are passed to the processor responsible for the new region. To enable solving of the field equations on the grid efficient, a secondary temporary decomposition is used to partition the simulation space evenly among the processors. After computing charge deposition by the particles, grid point data are exchanged among the processors to allow processors to solve field equations in their secondary partitions. For computational efficiency, field/grid data on the border of partitions are replicated on the neighboring processor to avoid frequent off-processor references.

As in [60], we perform a Beam-Plasma instability experiment in which a weak low density electron beam is injected into a stationary (yet mobile) background plasma of high density, driving plasma waves to instability. Experiments such as this can be used to verify plasma theories and to study the time evolution of macroscopic quantities such as potential and velocity distributions. Figure 3.2 shows an overview of the organization of the simulation program.

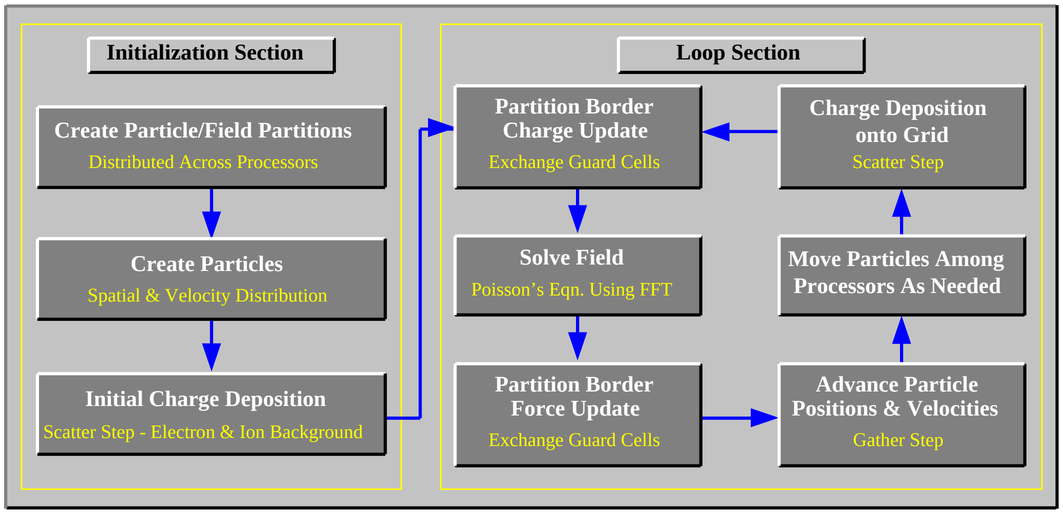
Initialization Section
Create Particle/Field Partitions
Distributed Across Processors
Create Particles
Spatial & Velocity Distribution
Initial Charge Deposition
Scatter Step - Electron & Ion Background
Loop Section
Partition Border Charge Update
Exchange Guard Cells
Charge Deposition onto Grid
Scatter Step
Solve Field
Poisson's Eqn. Using FFT
Move Particles Among Processors As Needed
Partition Border Force Update
Exchange Guard Cells
Advance Particle Positions & Velocities
Gather Step

Figure 3.2: Plasma PIC Computation Loop Overview. Reprinted with permission from [59] *under ©Copyright 1995 by ACM Inc.*

3.2.2 Plasma Simulation on NOWs Using a Distributed Grid

Norton et al [60] describe an object oriented implementation of the plasma simulation for distributed memory machines based on the GCPIC method. Their implementation partitions both the particles and the field data among the processors. As particles advance in space, they need to be redistributed to appropriate partitions. Redistribution of particles is achieved by a series of synchronous steps in which particles are passed from one neighbor processor to the next until they reach their destinations. Their implementation runs efficiently on parallel machines like the IBM SP2 and the Intel Paragon, which provide fast communication between the processors.

Our first attempt to implement the simulation on NOWs using the BSP model was to replace the message passing communication primitives with the communication and synchronization primitives from the Oxford BSP Library. No attempt was made to change either the data distribution or the simulation routines to improve the communication and synchronization performance of the implementation. This version was tested on a network of Sun Sparc workstations. As can be seen from Table 3.1, this implementation does not scale well. Increasing the number of processors from 4 to 8 reduces the computation time per processor (user time) by approximately 30%. However, the wall clock time for 8 processors is approximately

Number of Processors	20K Particles		
	Real Time	User Time	System Time
4	373.8	128.1	41.6
8	775.7	91.1	130.9

Table 3.1: Execution Times of the Plasma Simulation (Distributed Grid version) on a NOW (All times shown are in seconds).

twice that of 4 processors. These results confirmed our intuition: the distributed grid implementation is not suitable for the range of BSP parameter values that characterize NOWs.

It is important to understand the reasons for the poor performance of distributed grid implementation of plasma simulation on NOWs, since they are relevant to many other applications as well. According to the measurements on the IBM SP2 by Norton et al [60], advancing the particle positions and velocities accounts for about 90% of the overall simulation time and does not require inter-processor communication. Thus, the volume of the communication (total number of bytes delivered by the communication network) does not appear to be a problem even after taking into account the low bandwidth of the Ethernet. This relatively low volume of communication is spread over a large number of short messages, giving rise to a large number of communication calls and supersteps in the initial implementation on NOWs. This affects the performance in two ways. First, each communication call contributes an additional delay equivalent to the latency of the communication network. Second, every additional superstep contributes a synchronization point and adds synchronization delay to the overall execution time.

To better fit the BSP model, the distributed grid implementation can be improved by reorganizing the data distribution. Interprocessor communication can be reduced if all the interactions between the particle and the field data are local to each processor. Since the bulk of the work involves computations on particle data, the particles must remain partitioned among the processors to support workload sharing. However, the grid can be replicated on each processor allowing the particle/field interactions to remain local. Replicating the grid requires maintaining consistency of the grid data across all the processors. This is achieved by merging the modified copies of the grid from individual processors. All the processors send charge depositions to their local grid to one processor. This processor combines the individual charge depositions and broadcasts the final grid to all the processors.

3.3 Plasma Simulation Using a Replicated Grid

P. C. Liewer et al [41] implemented the plasma simulation on a 32 node JPL Mark III hypercube. The particle computations were done in parallel; each processor had a copy of the field data. Thus their implementation used a replicated grid, as a first step towards parallelizing the particle simulation codes. Our decision to use a replicated grid comes from an analysis of plasma simulation using the BSP model on NOWs. Replicating the field data on all processors has two positive effects:(i) it eliminates communication caused by particle-field interactions and (ii) it groups together communication needed to update field information on each processor. In addition, it avoids the need for a secondary decomposition and allows field computation to be performed locally on each processor thereby eliminating communication associated with exchange of grid points in the secondary decomposition.

There are other sources of execution inefficiency on a network of workstations. Parallel computer implementation [59] redistributes particles in a series of synchronous steps in which each processor exchanges particles with its neighbors. Such synchronization steps are very expensive on NOWs. In our current approach, the processors maintain a set of buffers, one for each processor. A particle which must be transferred to another processor is stored in a buffer corresponding to its destination. When this buffer is full or when all particles are processed, each processor sends its non-empty buffers directly to their destinations in a single superstep. Finally, replicating the grid allows for total elimination of particle redistribution. The evaluation of efficiency of such a solution versus the solution with buffered particle redistribution is given in the following section.

3.4 Analyzing BSP Implementation of Plasma Simulation

As the particles move in space, the region of the grid to which they deposit charge varies. Based on this criterion there are two approaches to organizing the particle data among the processors. In one approach (*scheme 1*) we redistribute particles after each advance operation, so that charges of the particles assigned to a processor are always deposited to the same portion of the grid based on the particle partitioning (approximately about $1/p$ of the grid in size). Only these portions of the grid from each processor need to be combined to construct the complete globally consistent grid. This also improves the cache performance of the simulation since only this portion of the grid needs to remain in the cache on each processor. However, redistributing the particles may cause the load to become unbalanced. In the other approach (*scheme 2*), we allow the particles to remain on the same processor throughout the simulation. This avoids the cost of redistributing the particles while maintaining the initial load balance. Since particles can now deposit charge anywhere, constructing the globally consistent grid requires combining the whole grid data from each processor. In addition, cache performance may also suffer during the charge deposition stage.

The total cost of a BSP implementation is the sum of the computation, communication and synchronization costs. In the plasma simulation, computation consists of advancing the position and velocity of the particles, depositing charge from each particle onto the grid, and solving the field equations. Field equations are solved by a sequential FFT with a complexity of $O(N_g \log(N_g))$, where N_g is the number of grid points. The charge deposition and particle advance operations are performed for each particle in constant time. Hence, assuming ideal load balance, the complexity of this part of the computation is $O(\frac{N_p}{p})$, where N_p is the number of particles in the simulation and p is the number of processors. Another computational step adds together the charge depositions contributed by the individual processors. In the first scheme, charge depositions to the grid points in the guard columns must be added to the corresponding grid partition. For the 2D simulations discussed in this paper, the height of the guard columns is approximately $\sqrt{N_g}$. Assuming that there are c_w overlapping guard columns per processor, the cost of summation is $(n-1)c_w\sqrt{N_g}$. In our implementation $c_w = 3$. In the second scheme, entire grids are added during the parallel gather operation, so the associated cost is $N_g \log(p)$.

To analyze communication costs, let c_{pm} be the average fraction of particles that cross an arbitrary selected plane perpendicular to the x-axis of the space in one simulation step. c_{pm} is dependent on the characteristics of the simulation (the speed distribution of the particles). With p processors, the average fraction of particles that change partitions is $c_{pm}p$. Let c_g be the size of data associated with a grid point and c_p be the size of data associated with a particle. Let $k = \frac{N_p}{N_g}$ be the average number of particles per cell.

In the first scheme, we use a linear gather for combining the grid data and a linear broadcast to return the grid to the processors. Particles are sent directly to the destination processor in one superstep. The communication cost in this case is

$$g_0(p-1)c_g(\frac{N_g}{p} + c_w\sqrt{N_g} + N_g) + g_0 N_p c_p c_{pm} p.$$

For the second scheme, there is no particle redistribution and the only cost is that of gathering the grid data from each processor into the global grid and broadcasting the resulting grid to all processors. We use logarithmic gather to combine the grid data, because summation of the data from individual processors can be done on the nodes of the tree in parallel. We use linear broadcast for sending the final grid to the processors. Following the analysis of the logarithmic and linear broadcasts in the previous section, the communication cost is $2g_0(N_g c_g)(p-1)$.

By adding the computation and communication costs derived above with the synchronization cost, we obtain the cost function for the first scheme (which uses particle redistribution):

$$\begin{aligned} T_1 \;=\;& c_1\frac{N_p}{p} + c_2 N_g \log(N_g) + c_w\sqrt{N_g}(p-1) + g_0(\frac{N_g}{p} + c_w\sqrt{N_g} + N_g)c_g(p-1) \\ &+ g_0 N_p c_p c_{pm} p + 3L_0\log(p) \qquad (3.3) \end{aligned}$$

where c_1, c_2 represent operation counts for charge deposition/force computation and field computation, respectively. Similarly, the cost function for the second scheme is

$$\begin{aligned} T_2 \;=\;& c_1\frac{N_p}{p} + c_2 N_g \log(N_g) + N_g\log(p) + 2g_0 N_g c_g(p-1) + L_0\log^2(p) \\ &+ L_0\log(p) \qquad (3.4) \end{aligned}$$

No of Proc	20K Particles			300K Particles			3.5M Particles		
	Real Time	User Time	System Time	Real Time	User Time	System Time	Real Time	User Time	System Time
4	510	143	9	5959	1919	32	68075	24352	232
8	410	102	20	3731	1036	77	27318	10225	322
16	504	92	53	3155	686	182	25619	5835	711

Table 3.2: Execution Times of Plasma Simulation (Replicated Grid version, with no particle redistribution) on a NOW (All times shown are in seconds).

Particle redistribution is more efficient than broadcasting the whole grid when (3.4) exceeds (3.3). That is when

$$\begin{aligned} N_g \log(p) + L_0 \log^2(p) + g_0 N_g c_g (p-1) &> g_0\left(\frac{N_g}{p} + c_w\sqrt{N_g}\right)c_g(p-1) \\ &+ c_w\sqrt{N_g}(p-1) + g_0 N_p c_p c_{pm} p + 2L_0\log(p) \end{aligned} \tag{3.5}$$

For large simulations using a large number of processors, the above inequality can be approximated by $N_g c_g > N_p c_p c_{pm}$. That is, redistributing particles is advantageous when the size of the grid data that needs to be shared by the processors is larger than the size of the particle data that needs to be exchanged among the processors. In our simulations, $c_g = 1, c_p = 4$ and c_{pm} ranges from 0.002 for 300,000 particles to 0.0009 for 3.5 million particles. With this data, the above inequality can also be written as $k < 130$ for 300,000 particles and $k < 280$ for 3.5 million particles. The value of k usually grows faster with the growth of the problem size than the value of c_{pm} decreases. Hence, the solution without particle redistribution should become more efficient than the alternative for very large simulations. Figure 3.3 shows a plot of the cost function for this solution. The values of the coefficients $c_1 = 458$ and $c_2 = 33$ have been estimated from theoretical analysis of the algorithms and experimental data. The plot indicates that the execution time continues to decrease up to 20 processors, even though the improvement may not be significant after 16 processors. In a synchronous application, computation proceeds at the speed of the slowest machine. As the number of processors increases, the

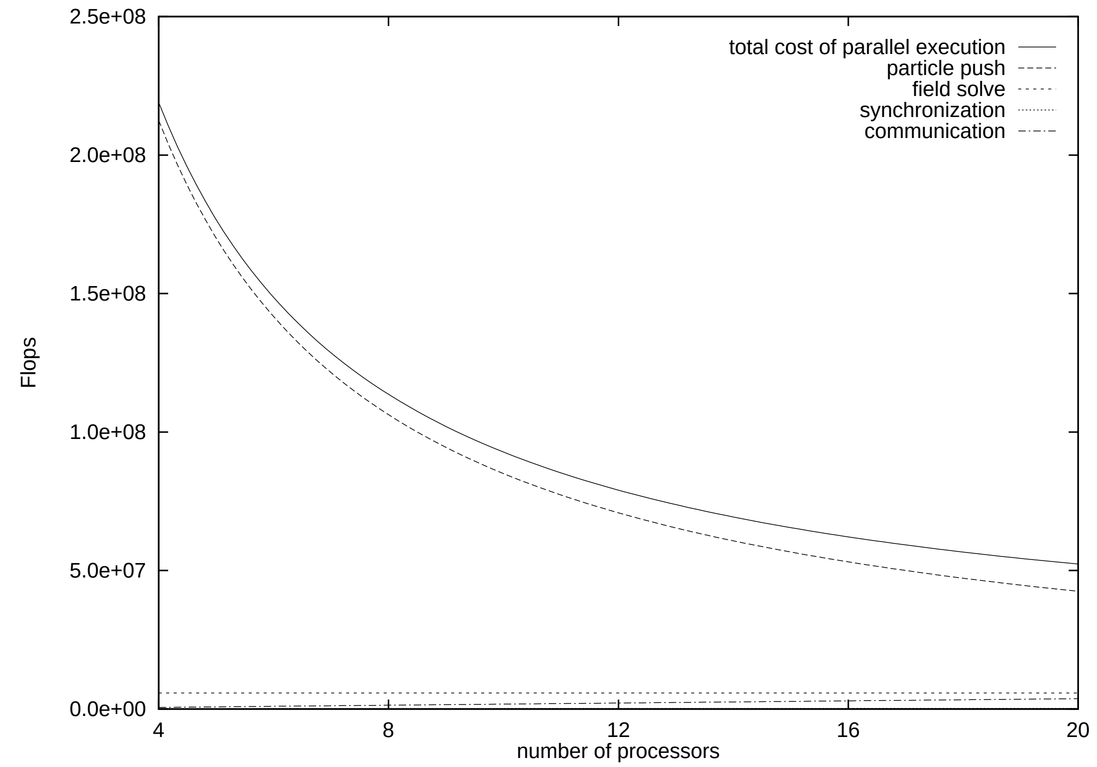

Figure 3.3: Cost function for plasma simulation using no particle redistribution. The simulation uses 3571712 particles and a grid of size 32768 ($k = 109$).

variation in the load (due to other users) also increases. The mean value of the random variable that represents the execution rate of the slowest machine decreases with increasing number of processors. This implies that the performance of the algorithm in a heterogeneous network will decrease statistically based on the overall system load. This explains why the observed execution times from Table 3.2 are larger than the theoretical cost function values of Figure 3.3. However, the trends in execution times shown in Table 3.2 correspond well with the cost function plot.

Table 3.2 shows an interesting phenomenon. For the case of 3.5 million particles, the simulation exhibits superlinear speedups as we increase the number of processors from 4 to 8. This phenomenon can be explained by the effects of the cache size on program execution time. For 8 processors program data fits in the system cache which reduces computation time due to improved memory access. Reducing program data below this size does not result in further improvements in memory access. When using 16 processors, decrease in computation time of local partition is offset by increased communication cost, resulting in significantly less increase in speedup from 8 to 16 processors.

3.5 Parallel Finite Element Computations on NOWs

Finite element methods are highly computationally intensive and parallel implementations are important for solving large problems in a reasonable time. Networks of general purpose workstations are widely used for scientific and engineering computations. Workstations' computing power can be utilized for the parallel computations if the parallel implementation is of sufficient efficiency. The main obstacle is the relatively high latency of the interconnecting network and sharing of the network bandwidth with other users.

As an example, we consider finite element modeling of a bioartificial artery. The techniques are applicable to all parallel finite element computations, including solution to problems arising in optimization and control of chemical and biological processes.

3.5.1 Overview of Finite Element Modeling of Bioartificial Artery

The model of tissue-equivalents was developed by Barocas and Tranquillo to study the compaction of collagen tubes by smooth-muscle cells (SMC). We present a brief overview of their model based on [7]. The cells exert a contractile stress on the surrounding collagen network, causing the gel to compact by exudation of water. Modeling this process is complicated by the cells' sensitivity to their surroundings: the cells will change their behavior in response to alignment of the collagen fibrils (which results for inhomogeneous strain) or increased stress in the network. Barocas and Tranquillo's model accounts for these phenomena as well as the viscoelastic behavior of the gel.

The model leads to a PDE system describing the mechanics of the fibril network and interstitial solution phases comprising the gel, cell division and movement within the network, and the forces exerted by the cells on the network. Defining the network viscoelastic stress (s), network velocity (v), solution phase pressure (p), network volume fraction (q), and cell concentration (n), the governing mechanical force balances for the network and solution phases can be written as

$$\nabla \cdot \left[\theta \left(\underline{\sigma} + n\underline{x}\right) - p\underline{I}\right] = 0 \tag{3.6}$$

$$\nabla \cdot \left(\frac{1-\theta}{\theta}\nabla p - \psi \underline{v}\right) = 0 \tag{3.7}$$

and the governing mass and species conservation equations as

$$\dot{\theta} + \theta \left(\nabla \cdot \underline{v}\right) = 0 \tag{3.8}$$

$$\dot{n} + n \left(\nabla \cdot \underline{v}\right) = \nabla \cdot \left(\underline{D} \cdot \nabla n\right) + kn \tag{3.9}$$

where material derivatives with respect to v are indicated. k is a constant measuring the cell division rate, and the tensors s and D describe the cell traction stress and cell migration; for certain isotropic systems, these tensors are just scalar multiples of the identity tensors, but for typical anisotropic systems (arising as a result of inhomogeneous deformation) the cells exert stress and migrate preferentially in the direction of network alignment. The details of modeling cell and network orientation have been described [6] and involve defining a local network orientation tensor in terms of an integral of the local strain tensor, and taking the cell orientation tensor to be a simple functional of that for the network.

Barocas and Tranquillo's numerical formulation is based on a mixed finite element method using piecewise bilinear pressure, piecewise biquadratic displacement, cell concentration, network concentration, and discontinuous piecewise biquadratic stress. Their formulation allows efficient and accurate solution of the model equations. Their solver performs temporal discretization using DASPK [63], and solves the large matrix problem using preconditioned GMRES [63] with an Uzawa-type preconditioner.

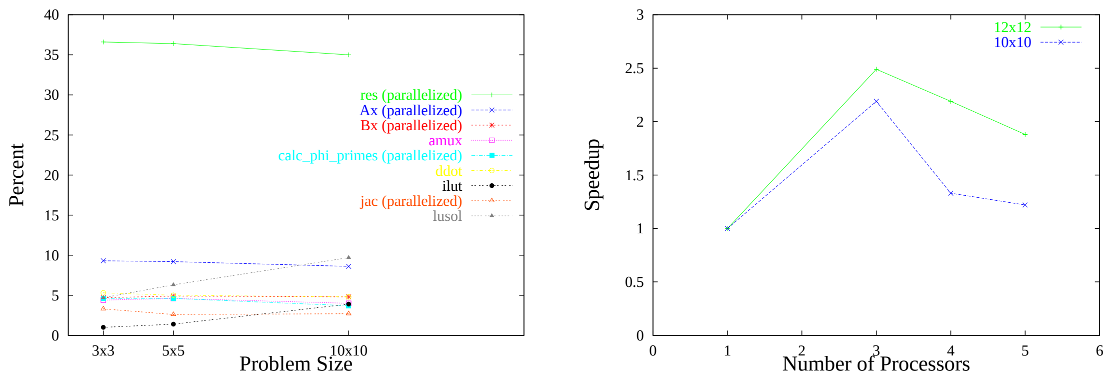

Figure 3.4: BSP implementation of modeling of bioartificial artery. (a) Execution profile of the routines (b) Speedups obtained

3.5.2 BSP Implementation of Finite Element Modeling of Bioartificial Artery

The parallel implementation using the BSP model performs elemental computations in parallel whenever possible. This is achieved by parallelizing loops that perform the elemental computations. The execution profile of the routines that are computationally significant is shown in figure 3.4(a). A brief description of the routines is given here.

res :

Calculates the residual equations $F(t, y, y')$ defined by the finite element discretization of the spatial problem. y is the vector of solution values, containing the variables (stress, position, cell concentration, collagen volume fraction, and pressure); y' is the vector of material derivatives of y. The subroutine DASPK calls res to determine whether the values of y and y' are acceptable for a time step. For a finite element mesh with N elements, the routine requires $O(N)$ operations.

calc_phi_primes :

Since the finite element mesh is not composed of perfect squares, the integrals on each element are solved by transforming the element to a unit square. This routine calculates the derivatives of functions based on the transformation. It requires $O(1)$ operations, but it must be called for every element.

jac :

Calculates the approximate jacobian that is used as a preconditioner for the iterative solution of the full jacobian (which is not needed). Like res, this routine operates by evaluating integrals on each element and summing, thus requiring $O(N)$ operations.

ilut :

The incomplete LU factorization of jacobian preconditioner blocks. Ilut is computationally expensive, requiring $O(N^2)$ operations.

lusol :

This is the solver for the above. It requires $O(N)$ operations.

amux :

Routine to multiply a matrix (in sparse row format) and a vector. Its complexity is proportional to the number of nonzeros in the matrix.

Ax and Bx :

These routines calculate the product between sections of the Jacobian matrix and a vector x. Since this product can be defined in terms of integrals over elements, the calculation requires $O(N)$ operations.

The constants in the O notation in all these computations are rather large, about 100 to 1000. For the problem sizes shown in Figure 3.4(a), calculation of the residuals is the computationally most significant and accounts for about 35% of the overall execution time. However, Ilut, with computational complexity quadratic in the number of elements, becomes more and more significant at higher problem sizes.

The figure also shows the routines that have been parallelized. As can be seen from the plot, about 60 to 70% of the sequential code is parallelized. Consequently, the maximum theoretical speedup that can be achieved from this computation as given by Amdahl's Law [1] is approximately 3. Our BSP implementation achieves a maximum speedup of about 2.5, which is obtained with 3 processors.

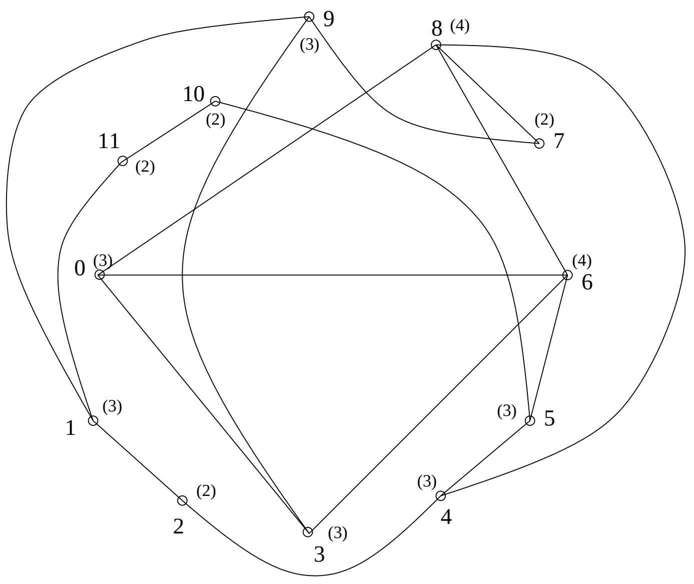

Figure 3.5: A sample graph with 12 vertices. Degree of each vertex is shown in parentheses

3.6 Finding the Maximum Independent Set of a Graph

A set of vertices in a graph is said to be an *independent set* if no two vertices in the set are adjacent [23]. A *maximal independent set* is an independent set that is not a subset of any other independent set. A graph, in general, has many maximal independent sets. In the maximum independent set problem, we want to find a maximal independent set with the largest number of vertices. Finding the maximum independent set has applications in communication theory. For example, if the vertices represent possible code words and edges connect code words that are close to each other and hence can be confused with each other, then finding the largest set of code words for reliable communication is the problem of finding a maximal independent set with the largest number of vertices.

To find a maximal independent set in a graph G, we start with a vertex v of G in the set. We add more vertices to this set, selecting at each stage a vertex that is not adjacent to any of the vertices already in the set. This procedure will ultimately produce a maximal independent set. In order to find a maximal independent set with the largest number of vertices, we find all the maximal independent sets using a recursive depth first search with backtracking [32]. To reduce search time, heuristics are used to prune the search space. The number of subgraphs searched at any level

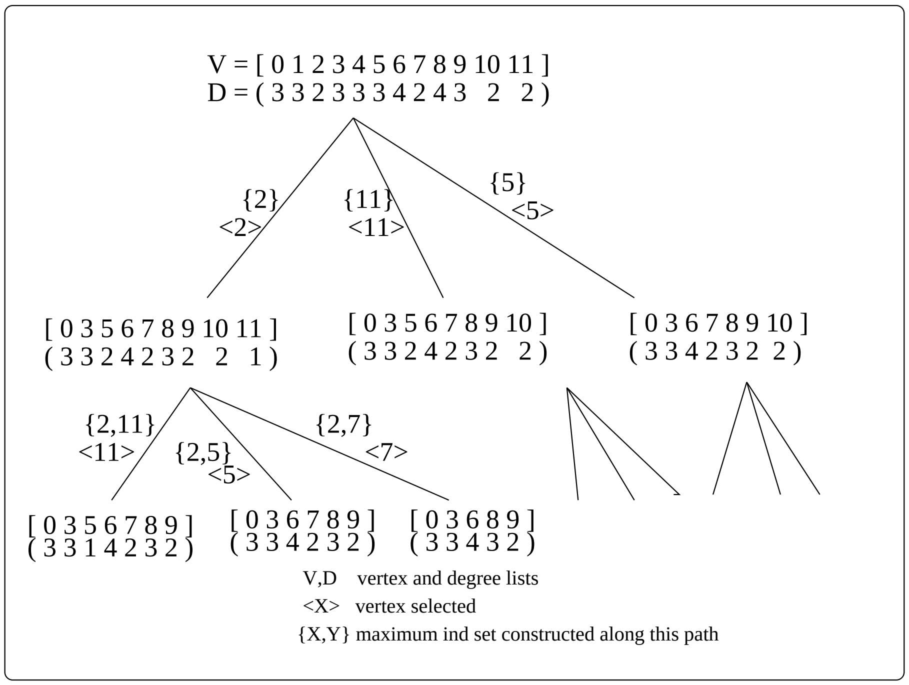

Figure 3.6: Search tree for the sample graph shown in Figure 3.5

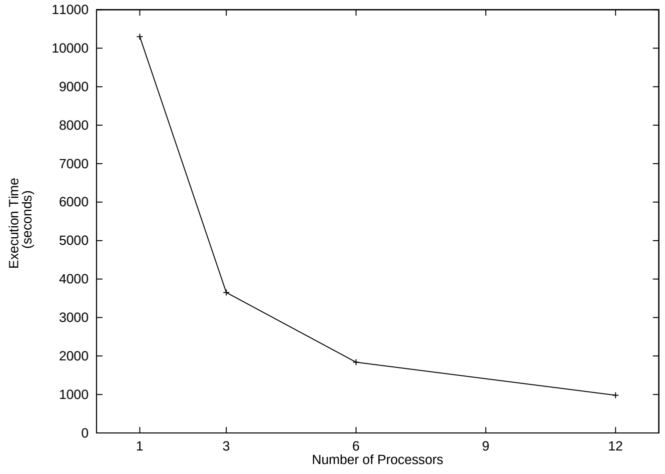

Figure 3.7: Plot showing speedups of the BSP implementation of the maximum independent set problem. All times shown are wall-clock times in seconds.

of recursion is determined from prior knowledge obtained from a large database of maximum independent sets found for various graphs and their search paths. The recursion is terminated when the size of the graph to be searched is too small to produce an independent set larger than the maximum independent set found so far.

Connectivity information for the vertices of the original graph is represented by an adjacency matrix. To conserve memory, no explicit representation of the graph is maintained. Instead, the connectivity information of the original graph is used to search through a virtual graph. The virtual graph is represented by two data structures: a list of all vertices in the graph and a list of their degrees. The degrees of the vertices in the graph are determined by referring to the adjacency matrix of the original graph. At any given stage, we select the vertex with the minimum degree and add it to the independent set being constructed. A new graph is formed by deleting the selected vertex and all vertices adjacent to it from the current graph. A new list of vertices is constructed to represent this graph. The list of vertices for the new graph is obtained by deleting the selected vertex and all vertices adjacent to it in the current graph from the current list of vertices. Once the list of vertices is constructed for the new graph, the list of degrees of the vertices is constructed by using the adjacency matrix. The new vertex list and the degree list representing the new graph are passed down to the recursive search function. Figure 3.5 shows a sample graph with 12 vertices and Figure 3.6 shows the corresponding search tree. Each node of the search tree corresponds to a level in the recursive search. Each path from the root to a leaf node results in the construction of an independent set. For each search, Figure 3.6 shows the vertex selected and the resulting subgraph to be searched.

In the parallel implementation of the search procedure using BSP, the adjacency matrix is replicated at all processors and the processors share the search space. The computation superstep involves each processor searching a subgraph. In the communication superstep, the processors exchange information on the maximal independent set found so far. This information is used to prune the search space in subsequent computation steps. Figure 3.7 shows a plot of the speedups obtained with the BSP implementation of the graph search.

3.6.1 Performance of the Graph Search Algorithm

Consider a graph with n vertices and a probability of d for any two vertices to be connected. The adjacency matrix is replicated at each processor. To conserve memory, the adjacency matrix is represented as bit patterns. If W is the word size on the host machine, then the storage requirements for the adjacency matrix is n^2/W. At each level of the recursive search, a vertex with minimum degree is added to the independent set and all vertices that are adjacent to it are deleted to form the virtual graph at the next level. Thus the number of vertices of the virtual graph to be searched decreases exponentially at a rate of $1-d$. Since the adjacency matrix is replicated at each processor, each processor can construct the virtual graph locally. At a given level of the recursive search l, the number of vertices of the subgraph is approximately $n(1-d)^l$. All processors proceed through the search space until the number of vertices of the subgraph to be searched is below a predetermined threshold L. Below this level, each subgraph is searched by one of the processors and all processors exchange information on the size of the maximum independent set found so far and its search path. Since the size of maximum independent set and the search path is usually very small, the cost of communicating these data can be considered to be constant. Thus the communication requirements of the graph search algorithm are quite low. However, since the adjacency matrix information is replicated at each processor, this algorithm does not scale. An improved graph search algorithm that partitions the adjacency matrix among the processors is presented in Section 7.4.1.

3.7 Summary

In this chapter we considered parallel computations on a network of dedicated workstation using the BSP model. We characterized the network of workstations as a BSP computer. We used the BSP model to characterize the application of plasma simulation and gain insight into the scalability of the problem. Using BSP analysis techniques, we tuned the application for improved performance on a network of workstations. We also analyzed the BSP implementation and characterized its behavior. We also presented an application of the BSP model to a finite element computation and a graph search problem. These examples illustrate that the BSP model can be used to model and implement a wide variety of parallel computations.

CHAPTER 4
PARALLEL COMPUTATIONS ON TRANSIENT PROCESSORS

In this chapter we consider the use of a network of nondedicated workstations for synchronous parallel computations. We describe the conditions under which additional computations[2] are allowed to run on a workstation. We characterize the workstations based on their availability and analyze the performance of additional parallel computations on the workstations. We use the results of the performance analysis to argue the need for making the parallel computations adaptive to the computing environment. We consider approaches to enable parallel applications adapt to the computing environment and analyze their performance.

4.1 Nondedicated Workstations as Transient Processors

Processors in a network of workstations (NOW) are often underutilized. Several studies indicate that a large number of workstations in a NOW are idle at any given time [15, 38, 52]. Arpaci et al [4] report that, although the set of idle machines changes over time, the total number of idle machines stays relatively constant. Our objective is to use the idle workstations to run additional jobs. There have been several systems that attempt to make use of idle workstations to execute sequential programs [19, 44, 58]. In systems using idle workstations, the additional computation is suspended when primary user activity is detected to avoid performance degradation for primary users. The additional computation is resumed when primary user activity ends and the workstation is idle. Since the workstations are available for use only when they are idle, they are referred to as *transient processors* [37]. A transition of the host processor from an available to a nonavailable state is referred to as a *transient failure.* When using a network of transient processors for parallel computation, each component process of the parallel application is assigned to a processor; the component process is scheduled when the host processor is idle

[2]Computations other than those belonging to the owner of the workstation.

and suspended when the processor is busy.

The impact of transient failures on sequential programs and long duration parallel programs with many independent tasks is analyzed by Kleinrock et al [37]. The rate of progress of long duration parallel programs with many independent tasks is proportional to the fraction of time the processor is idle. The impact of transient failures on frequently synchronizing parallel programs with relatively small amounts of computation between synchronizations is much more severe; if a single participating processor becomes unavailable, the entire parallel computation is delayed for the duration of the nonavailable period, making parallelism useless. As a result, synchronous parallel programs take much longer to execute on nondedicated networks of workstations. In the following sections we analyze the finishing time of frequently synchronizing parallel programs on a network of transient processors for the case of exponentially distributed available and nonavailable periods.

4.1.1 Approaches to Reduce the Impact of Transient Failures

Section 1.4.1 lists a number of approaches to reduce the impact of transient processor failures on synchronous parallel programs. These approaches try to reduce the impact of transient processor failures on program execution by replicating processes, computations and/or data to varying degrees. These approaches are classified based on the eagerness with which they replicate processes, computations or data. In this chapter, we consider the different approaches listed in Table 1.1. In the following sections, we analyze these approaches in more detail and compare the finishing times of a parallel program on transient processors under each approach.

4.2 Parallel Computations on Transient Processors

In the following analyses of finishing time of a parallel program under different schemes, we make the following assumptions:

- Transient processors alternate between an *available* and a *nonavailable* state. The lengths of the available (t_a) and nonavailable (t_n) periods are exponentially distributed random variables. They are mutually independent.

- The application is structured as a sequence of computation and communication steps.
- The application synchronizes frequently; in other words, the duration of a computation step in each component process is small compared to the mean available and nonavailable periods.
- The application program scales with the number of processors. That is, as the number of processors p increases, we can find a problem size n such that the parallel efficiency $e(n, p)$ is greater than a predetermined constant e_0.
- The application program is available on every processor in the network.

We explain these assumptions in more detail below and obtain expressions for the cost of data replication and the cost of process migration in terms of the program and network parameters. In addition to the above, schemes that depend on replication of computations may impose additional requirements on the computations. For example, the computation may not contain operations with side effects such as input/output operations.

4.2.1 Network

Mutka and Livny [52] made actual measurements of a network of transient processors. They developed models for the available and nonavailable period densities to fit these measurements. The model they used for the probability distribution function of available time was a 3-stage hyper-exponential distribution. For the probability distribution function of nonavailable time, they used a shifted 2-stage hyper-exponential distribution. Kleinrock et al, in their analysis of finishing time of a distributed computation on a network of transient processors [37], frequently assumed exponential distribution for the available and nonavailable periods in their examples. Using exponential distributions instead of hyper-exponential distributions will not affect the mean values of the results obtained, however the variance of the results will be lower than if hyper-exponential distributions were used.

Our measurements of available and nonavailable periods of workstations (shown in section 4.2.1.1) indicate that distribution of lengths of available and nonavailable periods have significant frequency of long periods. Since the finishing time of a computation step is determined by the maximum finishing times of individual processors, this results in a larger variance in execution time than with the assumption of an exponential distribution of available and nonavailable periods. To simplify the analysis, we too assume exponential distribution for the available and nonavailable periods of the processors. In subsequent sections, we analyze the finishing time of synchronous parallel programs using several approaches and conclude that only the adaptive replication scheme can support scalable computations on transient processors. In view of the measurements of Dinda et al [24] as well as our own measurements, for approaches based on smoothing, the variance of actual results will be larger than the variance derived with the assumption of an exponential distribution. Hence, the negative conclusion of our scalability analaysis will hold in this case. In case of our approach, which migrates computations on failed processors to available machines, actual results are not affected by frequency of long nonavailable periods. More frequent than expected long available periods on the other hand, will only improve performance by requiring fewer migrations. Hence, the positive conclusions of our scalability analysis of our approach hold in this case also.

4.2.1.1 Machine Characteristics

The available and nonavailable periods of workstations are monitored. Figure 4.1 shows the frequency distribution of lengths of available and nonavailable periods for two machines. As can be seen from the plots, the machines have large available periods which can be used for parallel computations. The large values of mean and standard deviation are due to long available periods during the nights when most machines are virtually unused. Figure 4.3 shows the minimum, mean and maximum values of lengths of nonavailable periods and the maximum to mean ratios of lengths of nonavailable periods for a few machines. As can be seen from the plot, the maximum length of nonavailable periods can be much higher than the mean length. There are a significantly large number of long available and nonavailable periods. Figure 4.2 shows the frequency distribution of available and nonavailable periods for a machine along with a curve with an exponential distribution to fit the

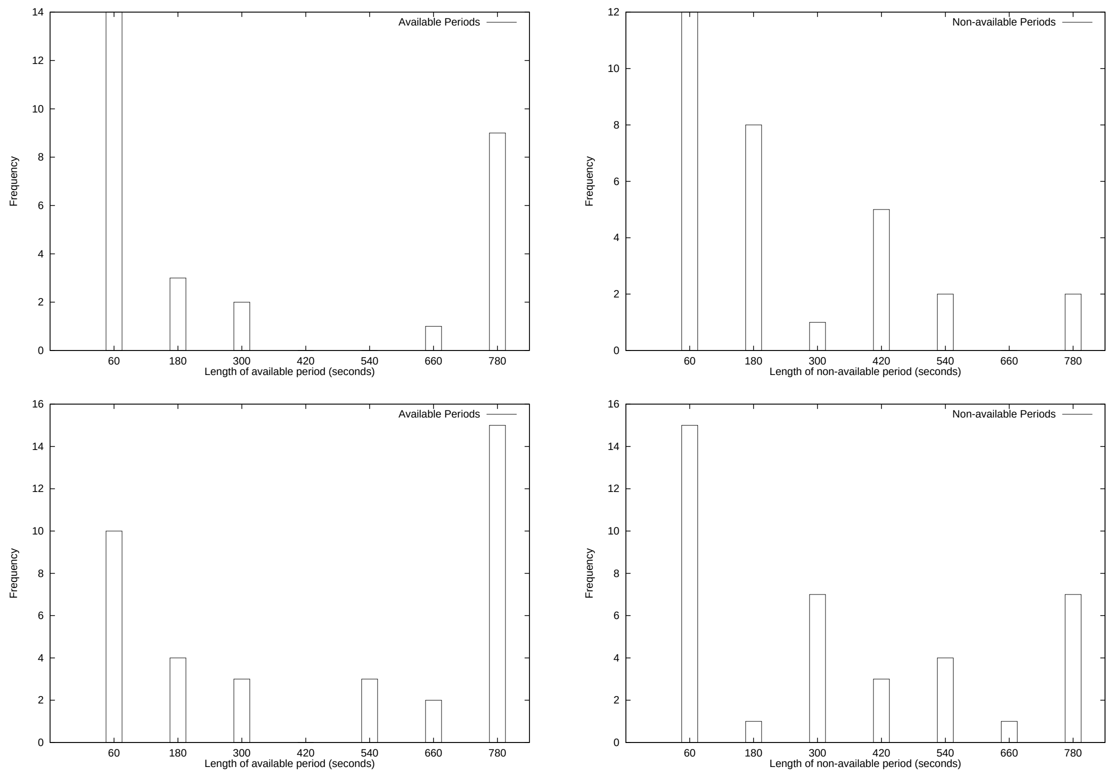

Figure 4.1: Frequency Distribution of lengths of available and nonavailable periods on two machines.

measurements. As can be seen from the plots, the tail portion does not fit well within exponential distribution. Dinda et al [24] collected detailed load measurements on a number of hosts. In their observations, they found that most machines had low mean loads, however load varied drastically. Desktop machines tended to have higher maximum loads relative to their mean loads. Our measurements with available and nonavailable periods exhibit a similar pattern. Their observations also show a high degree of self-similarity indicating that the variations in the available and nonavailable periods persist even over long periods. They observe that lessening the effects of load behavior by smoothing without migrating long-running tasks may not be effective. Our approach to adaptive parallelism is based on migrating tasks on unavailable processors.

Following Kleinrock et al, we assume a network of P identical processors. A processor alternates between a *nonavailable state* when it is being used by the owner, and an *available state* when it is idle. The lengths of nonavailable periods are in-

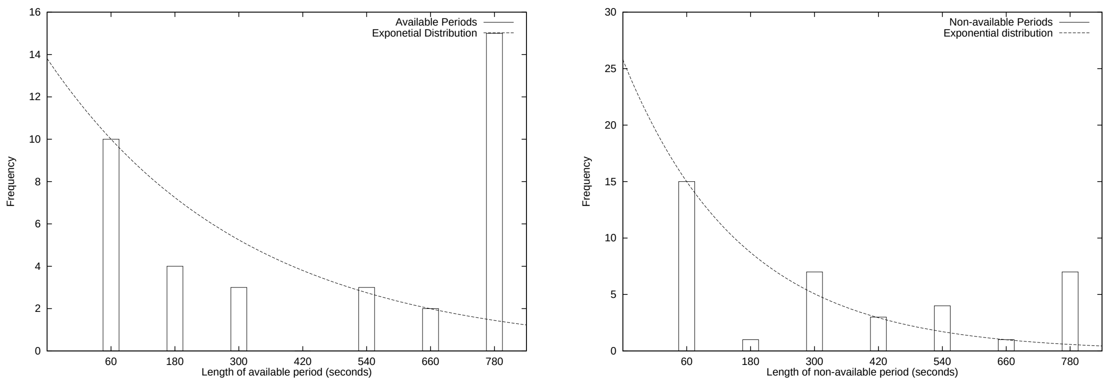

Figure 4.2: Frequency Distribution of lengths of available and nonavailable periods along with a curve with an exponential distribution. The tail does not fit with the exponential distribution

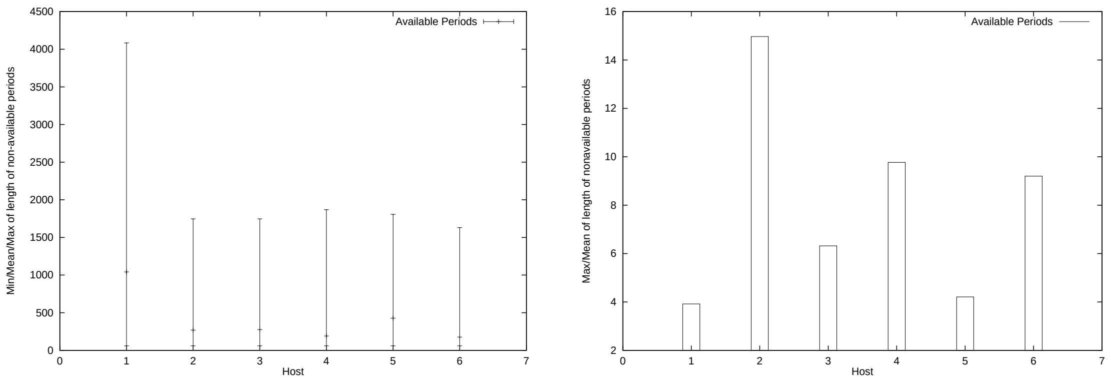

Figure 4.3: (a) Minimum, mean and maximum of lengths of nonavailable periods and (b) Maximum to mean ratios of lengths of nonavailable periods for a few machines

dependent and identically distributed (i.i.d.) random variables from an exponential distribution, $N(t) = \frac{1}{t_n} e^{-\frac{t}{t_n}}$, with mean t_n and variance ${\sigma_n}^2 = t_n$. Likewise, available periods are i.i.d. random variables from an exponential distribution $A(t) = \frac{1}{t_a} e^{-\frac{t}{t_a}}$. The available and nonavailable periods are mutually independent.

4.2.2 Application Program

The program is structured as a sequence of computation and communication steps. Assume a program with k computation and l communication steps. For simplicity, we assume that all the computation steps are identical. For a given problem of size n and a degree of parallelism p, let $T(n)$ be the total amount of computation, expressed as number of CPU seconds, done by all participating processors in each computation step and $C(n)$ be the size of the data communicated by each component process in each communication step. We drop the functional notation for the computation and communication when we refer to a given problem size.

4.2.2.1 Program Image

The program image consists of code and data segments. Let $I(n)$ be the size of the program image and I_{code} and I_{data} be the size of the code and data segments respectively, for a given problem of size n. The code segment is unaffected by the computation. The data segment is composed of two components: one that is common across all the processes and has size I_0 and another component of size I_d that is distributed over p component processes. Therefore, we have $I = I_{code} + I_{data}$ and $I_{data} = I_0 + I_d$. I_d consists of the data and results of the computation that can be parallelized, and I_0 consists of the data that is affected solely by the computation specific to each process, that is, the sequential part of the computation. The ratio of I_0/I_d is a measure of the scalability of the application, and for scalable applications the ratio approaches zero when the problem size grows to infinity. Hence for scalable computation and for large problem sizes, I_0/I_d is a small fraction, $I_0/I_d \ll 1$.

4.2.2.2 Computation State

For this analysis, we define the computation state of a component process P_i to be the collection of all the data that is necessary and sufficient in order for another component process P_j to reconstruct the state of P_i and execute (replicate) its computations. Let $S(n)$ be the size of the computation state of the program at the beginning of each computation step. Like the program data segment, the computation state consists of two components: a component of size S_0 that is common across all the component processes and another component of size S_d that is distributed across the p parallel components, $S = S_0 + S_d$. By definition, $S_0 \leq I_0$ and $S_d \leq I_d$. For scalable applications and large problem sizes, $S_0/S_d \ll 1$. Let g be the relative cost of communication compared to computation, expressed in terms of the number of local computation steps. The cost of communicating the computation state of a component process is gS_d/p and it represents the cost of data replication of a single component process.

4.2.2.3 Process Migration

We use the migration scheme of Condor [10] to migrate processes across machines. The process to be migrated writes its data segment at a checkpoint onto the disk and exits by invoking an exception. The program is restarted on a new host, by loading the checkpointed data from the disk. This scheme assumes the availability of the program code on each processor. This assumption is trivially true when using a network-wide file sever. The cost of migration under this scheme equals the cost of writing the data segment to the disk and reading it back. If g_d is the cost of disk access per unit data, then the cost of migrating a process is given by $T_m = g_d(I_0 + I_d/p) \approx g_d I_d/p$ for scalable applications and sufficiently large problem size.

4.3 Finishing Time of a Computation Step on Transient Processors

For the analyses, we define the completion of a step as the earliest time by which all the component processors have finished their local computation.

4.3.1 Finishing Time of a Step on a Single Transient Processor

When a single processor is used, there is no need to synchronize and the finishing time of a computation step depends on the number of failures during the execution of the step. Since the duration of a step T_s is small compared to the mean available period, the probability of occurrence of more than one transient failure during the execution of a step is negligibly small. For the analysis presented here, we assume the occurrence of at most one failure during the execution of a step. Hence, we are using an optimistic measure of the finishing time of the computation step in presence of transient failures.

Let $p(t)$ be the probability density function of the finishing time. Obviously, $p(t) = 0$ for $t < T_s$. For $t = T_s$, the step is executed without any failures and the probability is given by $\int_{T_s}^{\infty} \frac{1}{t_a} e^{-\frac{t}{t_a}}\, dt$. For $t > T_s$, the step is executed with failures of the node, so under the assumption of a single failure, the probability density of the distribution is given by $\frac{1}{t_n}(1 - e^{-\frac{T_s}{t_a}})e^{\frac{T_s}{t_n}}e^{-\frac{t}{t_n}}$.

The mean value of the finishing time is $T_s + (1 - e^{-\frac{T_s}{t_a}})\, t_n$, which can be approximated by $T_s(1 + \frac{t_n}{t_a})$ for $T_s \ll t_a$.

4.3.2 Finishing Time of a Step on p Transient Processors

Consider a parallel computation step with T units of work on p processors. Assuming uniform distribution of work among the participating processors, the computation step on each processor is of duration T_s, where $T_s = \frac{T}{p}$. Since all the processors synchronize at the end of the computation, the finishing time of the step is the maximum of the finishing times of the computation on individual processors. Let $F_p(t)$ be the cumulative distribution function of the finishing time of the step on p transient processors, so $F_p(t) = (F(t))^p$.

The probability density function $f_p(t)$ is the derivative of the distribution function $F_p(t)$, hence $f_p(t) \approx \frac{pD}{t_n}(e^{-\frac{t}{t_n}} - (p-1)De^{-\frac{2t}{t_n}})$ for $t > T_s$, where D stands for the constant expression $(1 - e^{-\frac{T}{t_a}})e^{\frac{T}{t_n}}$.

The mean finishing time of the computation step on p transient processors is approximately $\frac{T}{p} + T\frac{t_n}{t_a}$. We can see from the above expression that the mean finishing time of the parallel computation on p processors is worse than on a single processor whenever $t_n > t_a$.

4.3.3 Finishing Time on Transient Processors

The mean finishing time of a computation step when using p processors is $\frac{T}{p} + T\frac{t_n}{t_a}$. The mean finishing time of the program with k computation and l communication steps is, therefore

$$T_{par} = kT\left(\frac{1}{p} + \frac{t_n}{t_a}\right) + lgC \tag{4.1}$$

Since kT is the computation time of the sequential program, the parallel program performs worse than the sequential program whenever $t_n > t_a$.

4.4 Finishing Time of a Parallel Computation with Replicated Processes

4.4.1 Finishing Time of a Step Replicated on R Processes

Since the process is replicated on R distinct machines, the finishing time is the minimum of the finishing times of the R replicated copies. Let $f_R(t)$ be the probability density function and $F_R(t)$ the cumulative distribution function of the finishing time of a replicated process. Define D_r to be the constant expression $(1 - e^{-\frac{T_r}{t_a}})e^{\frac{T_r}{t_n}}$. For a single copy of the replicated process, the distribution function $F_{(R1)}(t)$ (on a transient processor) is given by $1 - D_r e^{-\frac{t}{t_n}}$ for $t > T_r$, $e^{-\frac{T_r}{t_a}}$ for $t = T_r$, and 0 for $t < T_r$. The distribution function for the replicated process $F_R(t)$ is the minimum of the R random variables, each with a distribution function $F_{(R1)}(t)$. The probability density function of the finishing time of the replicated process is given by $f_R(t) = \frac{d}{dt}(F_R(t)) = \frac{RD_r^R}{t_n}e^{-\frac{Rt}{t_n}}$. The mean finishing time of the replicated process is given by

$$T_r(1 - (1 - e^{-\frac{T_r}{t_a}})^R) + (1 - e^{-\frac{T_r}{t_a}})^R(T_r + \tfrac{t_n}{R})$$

4.4.2 Finishing Time of a Step With r Replicated Processes

In this section we analyze the finishing time of a computation step of duration T_r, with each process replicated on R machines. Since the total number of machines in the network is fixed (p), we now have $r = \frac{p}{R}$ parallel components. Further, the total work is constant, so $rT_r = T$, or $T_r = \frac{T}{r}$.

Each component has a finishing time whose density function and cumulative distribution function are as given above. The finishing time of the step is the maximum of the finishing times of the component processes. The distribution function of the finishing time of the step with r replicated processes is given by $F_{(rR)} = F_R(t)^r$. The probability density function of the finishing time of the step is obtained as $f_{(rR)}(t) \approx \frac{rRD_r^R}{t_n}\left(1 - (r-1)D_r^R e^{-\frac{Rt}{t_n}}\right)e^{-\frac{Rt}{t_n}}$. For a replication level ≥ 2 the mean finishing time is $T_r + r\left(\frac{T_r}{t_a}\right)^R \frac{t_n}{R}$.

The impact of the nonavailability of transient processors can be reduced by replicating each component process on more than one host machine. The finishing time of a computation step with r component processes when each process is replicated on R hosts is analyzed in the previous section.

4.4.3 Finishing Time of a Parallel Program using Full Process Replication

Consider the case in which additional processors are used for replication. Assume that each of the p components is replicated on R hosts. A computation step is complete when at least one of the R replicated processes of each component finishes its computation. Therefore, by the time a computation step is done, there could still be some replicas of a component that have not finished their computation because of an earlier transient failure of their host processors. To benefit from replication in the subsequent steps, replicated processes on currently available processors that have not yet finished their computations in the current step must be brought up to par with the leading process. To this end, the leading process communicates the computation state to lagging processes which use the data to update their own computation state. The cost of this operation depends on the size of the computation state and on the number of lagging processes.

The available and nonavailable periods form an *alternating renewal process* [21]. In the steady state, the probability of finding the processor in an available state is given by $t_a/(t_a + t_n)$. The average number of processors available in steady state, R', is given by $\sum_{i=0}^{R} i \begin{pmatrix} R \\ i \end{pmatrix} \left(\frac{t_a}{t_a+t_n}\right)^i \left(\frac{t_n}{t_a+t_n}\right)^{R-i} = R\left(\frac{t_a}{t_a+t_n}\right)$.

The cost of communicating the computation state to the lagging replica of a component process over k steps is $kg(R' - 1)S_d/p$. The cost of communicating the computation state over all the component processes depends on the properties of the communication medium. If there are independent communication channels between the processors, then communicating the computation state to the replicated processes can be done in parallel. In this section, we assume an Ethernet-based network which sequentializes communication of the computation state. Hence, the corresponding cost is $kg\,(R' - 1)\,S_d$.

Further, replication also affects the cost of communication internal to the running parallel program. When each component is replicated R times, the amount of communication increases at least R-fold. Hence, the mean finishing time of the parallel program is

$$\frac{kT}{p}\left(1+\frac{Tt_n}{t_a^2R'}\left(\frac{T}{pt_a}\right)^{R'-2}\right)+lgRC+kg(R'-1)S_d \tag{4.2}$$

For a scalable network, $g(R'-1)S_d \leq \frac{T}{p} < t_a$. This expression indicates that actively replicating processes is efficient and scalable only when the amount of communication and the size of computation state are relatively small.

4.5 Parallel Computations Using Failure-Recovery Scheme

An approach based on failure recovery can be used to improve the efficiency of synchronous parallel programs on transient processors. In this scheme, no processes are replicated. The scheme relies on the assumption that one of the processes is on a host owned by the user and, hence, not subject to the transient failures. We refer to this process as the master process. When a participating processor becomes unavailable, it communicates its computation state to the master process. The master process creates a new process and restores its state to that of the failed process so that the newly created process performs the necessary computations in place of the failed process. The new process is migrated to another host if one is available.

4.5.1 Finishing Time of a Parallel Program Using Failure-Recovery

In this scheme, failure recovery is always possible since a reliable (master) process is responsible for failure recovery. However, this also makes failure recovery strictly sequential and the finishing time depends on the number of failures. For each failure, the recovery time is the sum of the cost of communicating the computation state to the master process and the cost of executing the computations on behalf of the failed process. For large values of p and $T_s \ll t_a$, the mean number of processor failures is given by $p\frac{T_s}{t_a} = \frac{T}{t_a}$. Thus the mean number of failures increases with increasing problem size. The cost of communicating the computation state is gS_d/p for each failed process. In addition, the failed processes should be migrated to new hosts if available. Assuming the availability of a new host for each failed process, the overall finishing time will be $\left(\frac{T}{t_a}\right)(gS_d + T + g_d I_d)$.

The finishing time of a parallel program with k computation and l communication steps using the failure-recovery scheme based on lazy data replication and lazy computation replication is

$$\left(\frac{T}{t_a}\right)(kgS_d + kT + kg_d I_d) + lgC \tag{4.3}$$

Thus the scheme is not scalable as the number of failures and the cost of recovery increase with increasing problem size.

4.6 Parallel Computations Using Adaptive Replication

A variation of the failure-recovery scheme, proposed in [55], uses eager data replication and lazy replication of computations. In this scheme, each process is ready to act as a warm backup for one or more of its peers. This is achieved by eager replication of the computation state as explained below. At the beginning of each computation step, each process communicates its computation state to one or more of its peers, called the *backup processes*, before starting its own computations. In the event of the failure of the sender process (due to unavailability of the sender's host processor), one of the backup processes can replicate the computations of the failed process using the computation state received from that process. The participating processes are arranged in a logical ring topology and follow a protocol to ensure that the computations of the failed processes are performed in an orderly manner without duplication. The computation step is complete when all the computations are successfully completed by either the respective component processes or their backups. The number of backup processes assigned to each process is called the *replication level*, denoted by R. It is the number of processes at which the computation state of each process is replicated. Conversely, it is also the number of peer processes for which a process can act as a backup. It defines the maximum number of consecutive transient failures of processes, according to the order of the processes in the logical ring topology, that the data replication scheme can tolerate. Failure of more than R consecutive processes will force the processes to wait till one of the host processors recovers. This approach does not use any additional processors since no processes are replicated; computations are replicated only when a component process has failed. The investment is data replication whose main cost is the additional memory required for replication and the communication cost associated with sending the state of each subcomputation to peer processes. The cost of replicating the computation state can be reduced in some cases by incremental saving of the computation state. Incremental state saving has been used in discrete event simulation to reduce the cost and storage associated with restoring the state of a simulation object during rollback in optimistic protocols [67].

In this scheme, recovery of a failed process is performed by a peer process on an available processor. The recovered process initially runs on the host machine of the process that performed the recovery. This results in reduced parallelism which must be avoided if possible. A scheme to exploit adaptive parallelism, therefore, should provide for migrating the recovered processes to new available hosts. However, the scheme allows the parallel computation to proceed with the reduced parallelism if no new machines are available. In practice, the recovered process is migrated to a new available host as soon as one is available. The scheme thus allows the parallel computation to adjust to the number of machines currently available. We refer to this scheme as an *adaptive replication scheme.*

4.6.1 Performance of Adaptive Replication Scheme

For the purpose of this analysis, we assume that the computation state of a process is available as a consecutive array of bytes. We refer to this as the *primary data block.* The cost of data replication is the cost of communicating the primary data block to a peer process. The cost of data replication is linear in the size of the primary data block. However, since all the component processes attempt to communicate the primary data block in parallel, the cost of data replication depends on the characteristics of the communication medium. An Ethernet-based network has a fixed bandwidth that does not scale with the number of processors used. Such a network limits the scalability of the data replication scheme and may limit the amount of parallelism in a parallel application using data replication. If there are dedicated communication channels between the processors so that the network bandwidth scales with the number of processors used and the primary data blocks can be communicated in parallel, the data replication scheme scales as well. Communication media employing new technologies such as the ATM (Asynchronous Transfer Mode) allow a higher degree of parallelism thanks to their larger bandwidth.

To reduce the cost of data replication, the adaptive replication scheme seeks to overlap the communication associated with data replication with the local computation. Data replication can be done without significant overhead as long as the time taken for communicating the primary data block is less than the time of computation, under the assumption of overlapped communication and computation. If g is the relative cost of communication compared to local computation, we require that $RgS_d < T/p$ for a fixed bandwidth communication medium and $RgS_d < T$ for a communication medium whose bandwidth scales with the number of processors used. This limits the parallelism to $\frac{1}{Rg}(T/S_d)$ for a network with fixed bandwidth. Note that (T/S_d) is the number of operations on a unit size data item in the computation step. Therefore, data replication is advantageous for computation intensive parallel applications. Note that a similar situation exists in the case of the full process replication scheme, where each of the lagging processes must be updated with the computation state of the leading process. Since this update can occur only after the completion of the computations by the leading process, this data communication cannot be overlapped with the computation. Further, in the full process replication scheme, the computations of all but one of the replicated processes are not used.

The following analysis makes these notions more concrete.

4.6.1.1 Computation Dominant Applications

In this section we consider the performance of the adaptive replication scheme for applications in which the cost of data replication, expressed in O-notation, is no more than than the local computation. Hence, the finishing time of the step is within a constant factor of the local computation. Assuming a replication level of R, we distinguish three cases for the execution of a computation step: execution of the step with no transient failures, execution with more than R failures and execution with at most R failures. In the first case of failure free execution, execution time is given by $T_s = \frac{T}{P}$; the probability of no component failures is given by $(e^{-\frac{T_s}{t_a}})^p = e^{-\frac{T}{t_a}}$. In the second case the computation can proceed only after a time t_n, since the number of failures is more than the maximum number of failures that can be tolerated by the adaptive replication system. The execution time in this case is $\frac{T}{p}+t_n$ and the probability of failure of $R+1$ or more successive component processors is $p(\frac{T_s}{t_a})^{R+1} = \frac{1}{p^R}(\frac{T}{t_a})^{R+1}$. This is an optimistic measure since additional processes can fail during the recovery of other failed processes. In the third case execution time of the computation step depends on the number of successive processor failures. Since the maximum number of successive processor failures the adaptive replication system can tolerate is R, the worst-case execution time is given by $(R+1)\frac{T}{p}+T_m$, where $T_m = g_d I_d/p$ is the cost of migrating the failed process to a new available host. Note that since process recovery can proceed concurrently with migration of a recovered process, we charge for only one migration. The mean execution time of the computation step when using adaptive replication scheme is given by the weighted average of the three cases:

$$T_s = \frac{T}{p}e^{-\frac{T}{t_a}} + \left(\frac{T}{p}+t_n\right)\frac{1}{p^R}\left(\frac{T}{t_a}\right)^{R+1} + \left((R+1)\frac{T}{p}+T_m\right)\frac{T}{t_a}\left(1-\left(\frac{T/p}{t_a}\right)^R\right) \quad (4.4)$$

Large Computations

We are interested in scalability of large computations, defined as those for which the total computation in a step is comparable to or larger than the mean available period, we have the following expression for the mean execution time. Note that, even though T is not small compared to t_a, we can choose p such that $\frac{T}{p} \ll t_a$.

$$\begin{aligned} T_s =& \tfrac{T}{p}e^{-\frac{T}{t_a}} + \left(\tfrac{T}{p}+t_n\right)p\left(\tfrac{T/p}{t_a}\right)^{R+1} + \left(1-e^{-\frac{T}{t_a}} - p\left(\tfrac{T/p}{t_a}\right)^{R+1}\right)\left((R+1)\tfrac{T}{p}+T_m\right) \\ =& \tfrac{T}{p}\left(1+R\left(1-e^{-\frac{T}{t_a}} - p\left(\tfrac{T/p}{t_a}\right)\right)\right) + t_n p\left(\tfrac{T/p}{t_a}\right)^{R+1} + T_m\left(1-e^{-\frac{T}{t_a}} - p\left(\tfrac{T/p}{t_a}\right)^{R+1}\right) \end{aligned}$$

$$\leq \quad \tfrac{T}{p}(1+R) + t_n p \left(\tfrac{T/p}{t_a}\right)^{R+1} + T_m \left(1 - e^{-\frac{T}{t_a}} - p\left(\tfrac{T/p}{t_a}\right)^{R+1}\right)$$

From the above expression, the scheme is scalable for large computations as long as $p\left(\frac{T/p}{t_a}\right)^R \frac{t_n}{t_a} < (R+1)$ and $T_m \leq (R+1)\frac{T}{p}$ when data replication is scalable. These two constraints allow us to choose suitable values for the degree of parallelism p and the replication level R to maintain scalability. Since t_n and t_a are comparable, the two constraints are satisfied when we choose p and a corresponding value for R such $\frac{T}{t_a} < p \leq (R+1)\frac{T}{T_m}$. From this expression, we get a range of values for the degree of parallelism, p. This range is nonempty when $T_m < (R+1)t_a$. This expression reflects the architectural requirements for scalability of the adaptive replication scheme. Since component processes can fail in different groups, recovery and migration of processes in different groups can occur concurrently. Therefore, a parallel architecture with low migration cost and a scalable network can ensure scalable parallel computations using the scheme.

4.6.1.2 Data Replication Dominant Applications

We now consider applications in which the cost of data replication exceeds the local computation and hence the finishing time of the computation step is determined by data replication. Again, we analyze three cases introduced in the previous subsection. The execution time for the first case is given by gS_d. For the second case, the execution time is $gS_d + t_n$. Similarly, for the third case, the execution time is $gS_d + R\frac{T}{p} + T_m$, where T_m is the cost of migrating the failed process to an available host. The mean execution time is

$$gS_d + \frac{T}{p}\left(\frac{RT}{t_a}\right) + \frac{t_n}{p^R}\left(\frac{T}{t_a}\right)^{R+1} + T_m\left(\frac{T}{t_a}\right) \tag{4.5}$$

Finishing time of a parallel program with k computation and l communication steps is given by

$$kgS_d + \frac{kT}{p}\left(\frac{RT}{t_a}\right) + \frac{kt_n}{p^R}\left(\frac{T}{t_a}\right)^{R+1} + kT_m\left(\frac{T}{t_a}\right) + lgC \tag{4.6}$$

Since the cost of data replication exceeds the computation time, the scheme will not be scalable for data replication dominant applications for an Ethernet based communication medium. However, if the communication medium scales with the number of processors used (for example, with connections between every pair of processors), then data replication can be done in parallel by all the processors and this scheme will be scalable as long as $O(\frac{S_d}{T}) = O(1)$.

4.7 Summary

In this chapter we analyzed the finishing time of a parallel computation executing on a network of transient processors. We also analyzed the program execution time under various approaches including the adaptive replication scheme. Based on the analysis, we classified parallel computations into two categories: computation dominant and replication dominant. Computation dominant applications can be executed on nondedicated workstations without loss of efficiency. Classification of a parallel computation into these categories depends not only on the characteristics of the parallel computation, but also on the properties of the communication network. To achieve efficient computation of a wide variety of parallel applications on nondedicated workstations requires a network with low latency and high bandwidth. This is also the requirement for efficient parallel computations on dedicated workstations. Therefore, by investing more in the network connecting the workstations, we can achieve efficient parallel computation on nondedicated as well as dedicated workstations.

CHAPTER 5
ADAPTIVE PARALLELISM IN THE BULK-SYNCHRONOUS PARALLEL MODEL

In this chapter, we discuss the adaptive replication scheme within the framework of the Bulk-Synchronous Parallel model [56]. We describe the protocol used by the processes participating in the BSP computation for replication of data and for recovering the failed computations. We also describe the algorithm used by the participating processes.

5.1 Adaptive Replication Scheme

An overview of the adaptive replication scheme is given in the previous chapter. As explained in Section 4.6, the adaptive replication scheme relies on executing (replicating) the computations of a failed process on another participating processor to allow the parallel computation to proceed. Note that in a synchronous computation, the computation states of the participating processes are consistent with each other at the point of synchronization and therefore, by starting with the state of a failed process at the most recent synchronization point and executing its computations, we can recreate the state of the failed process and execute its computations. This allows the parallel computation to proceed in spite of the failure of some of the participating processes. This approach takes into account the nature of the computing environment in NOWs, in which the machine cycles are relatively inexpensive since we are mainly using idle machine cycles. Replicating the computations of failed processes is made possible by eagerly saving the *computation state*[3] (C_s) of each process on a peer process at the beginning of the computation step.

In our approach the computation state is eagerly replicated and the computations are replicated only as needed by the failure of one or more of the participating processes. Our approach can therefore be characterized as *eager replication of com-*

[3]The precise definition of what constitutes the computation state is given in the next chapter. Not all of the computation state needs to be replicated.

putation state and *lazy replication of computations.*

5.1.1 Assumptions

The adaptive replication scheme as described in the following sections assumes that one of the processes is on a host owned by the user and hence this process is immune to transient failures. We refer to this reliable process as the *master process.* The master process coordinates recovery from transient failures without replicating for any of the failed processes. It should be noted that the assumption of existence of a reliable process is not necessary for the correctness of the protocol. Using the standard techniques from distributed algorithms, synchronization can be achieved over the virtual ring regardless of transient failures. However, the master process is a convenient solution for a majority of applications, so we used it in this prototypical implementation of the system.

For the prototypical system implemented, we assume that supersteps that make use of replication contain computation only. This is not overly restrictive because, in the BSP model, data communicated in a superstep are guaranteed to be received at the destination process only by the end of the superstep and can only be used in the next superstep. Hence a superstep involving both computation and communication can always be split into a computation superstep and a communication superstep. This assumption greatly simplifies the design of the protocol for the recovery of failed processes. Further, the protocol assumes a reliable network, so a message that is sent by a process will always be received at the destination.

5.1.2 Protocol for Replication and Recovery

The participating processes other than the master process are organized into a logical ring topology in which each process has a predecessor and a successor. Recall that in our model based on BSP, computation consists of a sequence of synchronized computation and communication supersteps. In a computation superstep, each process in the ring communicates its computation state, C_s to one or more of its successors, called *backup processes*, before starting its own computation. Each process also receives the computation state from one or more of its predecessors.

When a process finishes with its computation, it sends a message indicating successful completion to each of its backup processes. The process then checks to see if it has received a message of completion from each of its predecessors whose computation state is replicated at this process. Not receiving a message in a short timeout period is interpreted as the failure of the predecessor. The process then creates new processes - one for each of the failed predecessors. The computation state of each new process is restored to that of the corresponding failed predecessor at the beginning of the computation step. To do that, the restoring process uses the computation state received from that predecessor. The process then performs synchronization for itself. Each of the newly created processes performs the computations on behalf of a failed predecessor and performs synchronization on its behalf to complete the computation step.

5.1.3 Process Migration

As explained above, computations of the failed processes are replicated by surviving processes which create new processes for this purpose. In general, such a newly created process assumes the identity of the corresponding predecessor and can continue participating in the parallel computation as a legitimate member. However, for the sake of better performance, this renewed process is migrated to a new host if one is available. We use a checkpoint based migration scheme to migrate the process to the new host machine. The migration scheme is explained in chapter 6.

5.1.4 Tolerating Multiple Failures

Replication of computations is made possible by eager replication of the computation state of a component process on one or more peer processes as explained above. Recall from Section 4.6 that the number of successors on which the computation state of a component process is replicated is referred to as the replication level (R). It is also the number of predecessors from which a process will receive the computation state. A process can therefore act as a backup to any of the R predecessors from which it receives the computation state. It is easy to see that the replication level defines the maximum number of adjacent process failures (according to the ordering of the processes in the logical ring topology) that the system can tolerate.

Simultaneous failure of more than R adjacent processes will force the processes to wait till one of the host processors recovers. A higher level of replication increases the probability of recovery from failures, but in networks with a fixed bandwidth, it also increases the overhead during normal (failure free) execution.

5.2 Adaptive Replication Algorithm

In this section we describe the algorithm for the adaptive replication scheme. Each of the participating processes follows the algorithm to replicate the data, perform local computation, detect the failure of predecessors in the logical ring topology, perform recovery for any failed predecessors, and finally to perform synchronization.

Consider a parallel application with p component processes. In a computation step, each component process performs some local computation on the local data (the computation state). In the adaptive replication scheme each component process communicates the computation state to one or more backup processes as determined by the replication protocol. Thus each process receives the computation state from each of the processes for which it is acting as a backup. In the normal execution, each process performs computations on its local data. When replicating for a predecessor, the process performs computations on the computation state data received from its predecessor.

Figure 5.1 illustrates the replication protocol for a replication level (explained below) of one. The adaptive replication algorithm is shown in figure 5.2. Each of the participating processes executes the adaptive replication algorithm in each superstep.

5.2.1 Data Replication

At the beginning of a computation step, each process communicates its computation state to its successor, which acts as the backup process. Each process also receives the computation state of its predecessor, which it saves as a backup copy.

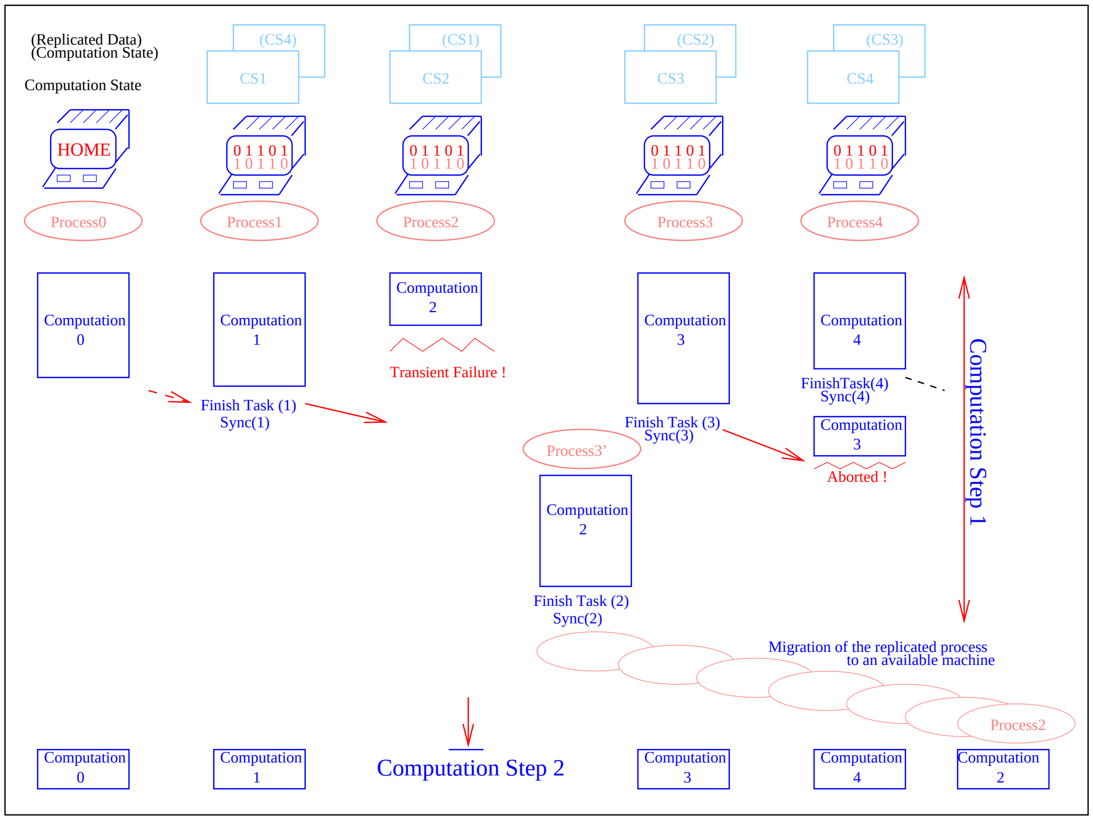

Figure 5.1: The Adaptive Replication Scheme illustrated for a replication level of one.

5.2.2 Primary Computation

Each process then performs local computation using its computation state. A process suspends itself anytime during the local computation if it receives a signal indicating that the host machine is unavailable. When a process completes its local computation, it sends a message indicating successful completion to its successor.

5.2.3 Secondary Computation

Upon successful completion of the primary computation stage, each process checks to see if its predecessor has successfully finished its computations. This verification is based on the receipt or absence of the successful completion message from the predecessor. A suitable timeout period may be applied before deciding that the predecessor has failed. When a participating process decides that its predecessor has failed, it creates a new process to replicate the computations of the failed predecessor. The new process restores the computation state of the failed predecessor using the state data it has received at the beginning of the computation step. It then performs the local computation on behalf of the predecessor. The child process aborts the computation any time it is informed by the parent that the predecessor has successfully finished its computations.

5.2.4 Conclusion

Each surviving process initiates synchronization for itself. Likewise, each of the newly created processes that replicated for a predecessor initiates synchronization on behalf of the predecessor. The processes then wait for the synchronization to complete. When the synchronization is complete, each of the surviving processes receives information on the list of processes that successfully completed synchronization. Based on this information, the surviving processes inform their child processes either to continue or to terminate. Upon hearing from the parent process to continue, the child process will assume the identity of the failed predecessor and will continue to participate in the parallel computation as a legitimate member.

5.3 Summary

In this chapter, we presented our approach to adaptive parallelism in the Bulk-Synchronous Parallel model. The synchronous nature of the BSP model simplifies the scheme by eliminating the need for a consistent checkpoint and by providing a convenient point for checking process failures. The protocol for replication and recovery assumes no knowledge of the transient failures and thus can be extended to deal with real processor failures. We also presented the replication algorithm used by the participating processes.

- **Data Replication**

  1. Communicate the computation state to the successor.
  2. Receive the computation state from the predecessor.

- **Primary Computation**

  1. Perform local computation.
  2. Send a *FinishTask* message to the successor indicating successful completion of local computation.

- **Secondary Computations**

  1. Verify if the predecessor has successfully finished its computation. Verification of successful completion of the predecessor is based on the receipt or absence of a *FinishTask* message from the predecessor.

     A suitable *timeout* period may be applied before checking for the message from the predecessor to account for differences in processor speed or delays in the communication network.
  2. If the predecessor has not finished its computation, then create a new (child) process to replicate the computations of the failed predecessor. The child process does the following:

     (a) Restore the computation state of the failed predecessor at the beginning of the computation state using the computation state received from the predecessor in step 1.

     (b) Perform the local computations on behalf of the predecessor.

     (c) Initiate synchronization on behalf of the predecessor.

     (d) Abort the computation of the predecessor and exit at any time if a signal indicating successful completion of the predecessor is recceived from the parent process.

Figure 5.2: Adaptive replication algorithm (continued on the next page)

- **Conclusion**

  The superstep is complete when synchronization is initiated on behalf of each participating process either by the original component process or by its successor.

  Each surviving process does the following:

  1. Wait for the successful completion of synchronization.
  2. If the child process which has replicated for the predecessor has successfully completed synchronization, inform the child process to continue; otherwise inform the child process to abort.

  The child process does the following:

  1. Wait for a signal from the parent to continue or to abort, based on whether or not synchronization by the child process is successful.
  2. Upon being informed by the parent to continue, the child process does the following:
     - Assume the identity of the predecessor and continue in place of the predecessor.
  3. Upon being informed by the parent to abort, the child process terminates.

Figure 5.3: Adaptive replication algorithm (continued from the previous page)

CHAPTER 6
DESIGN AND IMPLEMENTATION OF THE ADAPTIVE REPLICATION SYSTEM

In this chapter we discuss the design and implementation of the adaptive replication scheme (ARS). ARS was designed as a set of layers on top of an extended version of the Oxford BSP Library [48]. We describe the extensions to the library and the design of the runtime system consisting of the replication and user layers [57]. We also describe a monitor to check the status of the host machines.

6.1 The Computation State

Replicating the computations of a failed process is made possible by eagerly saving the *computation state* of each process on a peer process at the beginning of the computation step. In Section 4.2.2.2, we defined the computation state of a component process P_i as the collection of all the data that is necessary and sufficient in order for another component process to reconstruct the state of P_i. The goal is to be able to execute the computations of P_i. However, not all of the computation state needs to be replicated. In this section we refine the amount of computation state of a component process that needs to be replicated to enable recovery of the component process by a peer process.

We need to communicate only the part of the computation state that is distinct in each component process, since the common part is readily available at the process performing the recovery. The part of the computation state that is distinct in each component process is referred to as the *Specific System State*, SSS. That part of the computation state that is common across the processes is referred to as the *Common System State*, CSS. Thus the computation state of a component process is the sum of the specific and common system states. The specific system state needs to be saved on a peer process; the common system state needs to be saved only if it is modified in the current superstep. However, unlike the SSS, which must be saved on a backup process, the CSS can be checkpointed locally on each process. Thus the cost of data replication includes the cost of communicating the specific system state and the additional memory associated with the checkpointing of specific and common system states.

Computations in which it is possible to recompute some or all of the computation state can reduce the cost of data replication by specifying the code to recompute the state. We refer to this code as the *Recovery Function.* In those applications, execution of the recovery function is performed after restoring the computation state (specific and common system states) from the backup. The recovery function is also useful in applications whose computation state is not directly accessible. For example, in an application using a vendor supplied random number generator, the computation state may include the state of the random number generator which is encapsulated in the library and not directly accessible to the user. A recovery function using the function calls from the vendor library will be able to recreate the computation state in such a case. We have used the recovery function to successfully recompute the computation state in an application using such a random number generator. A recovery function is also used to recompute the computation state in a graph search algorithm discussed in Section 7.2.

6.2 Design of the Adaptive Replication System

The Adaptive Replication System is designed within the framework of the BSP model [70] and developed using the Oxford BSP Library [48, 49]. ARS consists of dynamic extensions to the Oxford BSP library and the adaptive replication scheme. The adaptive replication scheme is designed in two levels of abstraction: a *replication layer* and a *user layer.* The replication layer implements the functionality of the adaptive replication scheme including the protocol for recovery and replication, as a set of primitives. However, these primitives are not directly accessible to the applications; the functionality provided by the replication layer can be accessed only through the user layer. By designing the runtime support in two layers, we intend to insulate the applications from changes in the implementation. By implementing the replication layer for other architectures, we can maintain the portability of applications using our library.

```
/* Process Management */
bspstart(int argc, char* argv[], int maxprocs, int id, int nprocs);
bspfinish();
/* Synchronization */
bspsstep(int stepid);
bspsstep_end(int stepid);
/* Communication */
bspstore(int pid, char* from, char* to, int nbytes);
bspfetch(int pid, char* to, char* from, int nbytes);
```

Figure 6.1: The Oxford BSP Library for the C language

6.2.1 Extensions to the Oxford BSP Library

The Oxford BSP Library implements a simplified version of the Bulk-Synchronous Parallel model. Figure 6.1 shows the library functions for the C programming language. The Oxford BSP Library has been extended to provide dynamic process management and virtual synchronization. The extensions include the following features: the component processes can be terminated at any time; new processes can be created to join the computation; and component processes can perform synchronization for one another.

6.2.2 Implementation of the Protocol for Replication and Recovery

The participating processes other than the master process are organized into a logical ring topology in which each process has a predecessor and a successor. Each process in the ring acts as a *backup process* for its predecessor. Each process in the ring communicates its specific system state to its backup process before starting its own computations, where it is stored as a backup copy (SSS-BACKUP). Each process also saves the common system state as a local checkpoint (CSS-BACKUP). When a process finishes with its computations, it sends a message indicating successful completion to its backup process. The process then checks to see if it has received a message of completion from its predecessor whose computation state is replicated at this process. Not receiving a message in a short timeout period is interpreted as the failure of the predecessor. The backup process then creates a new process and restores the computation state of the new process to the computation state of the predecessor at the beginning of the computation step using the backup copies of the specific and local system states saved at this process and the recovery function. Restoring the computation state of the new process involves

- restoring the specific system state from the backup copy received from the predecessor (SSS-BACKUP),
- restoring the common system state from the local checkpoint (CSS-BACKUP), and
- executing the user supplied recovery function.

The newly created process executes the computation step on behalf of the failed predecessor and performs synchronization on its behalf to complete the computation step.

6.2.3 Adaptive Replication Scheme: Replication Layer

The replication layer implements the functionality of the adaptive replication scheme, including the protocol for replication and recovery. It provides the following functionality for a component process:

- Replicate the specific system state on the backup process as determined by the replication protocol.
- Checkpoint the common system state locally on the same process.
- Detect the failure of the process whose computation state is replicated on this process.
- Create a new process to execute the computations of a failed process. The new process created is a child of the process performing the recovery.
- Restore the computation state of the newly created process from the backup copies of the specific and local system states.
- Execute the recovery function supplied by the user.
- Perform synchronization on behalf of a failed process.

- Terminate lagging processes whose computations have been successfully replicated.
- Migrate the process to another available host.

The replication layer allows a process to detect and replicate for failed processes. However functionality of this layer is not directly accessible to the user, but only through the user layer.

6.2.4 Adaptive Replication Scheme: User Layer

The user layer provides the application programming interface (API) for the Adaptive Replication System. It includes primitives that transparently allow access to the functionality of the replication layer. The user layer provides the following constructs:

- Constructs to specify data to be replicated and memory management for the replication data.

  The construct `bsp_replication_data` (see Figure 6.2 for the full syntax) allows the user to specify static or dynamic storage for the replication data. The user can specify static or dynamic storage for replication data by explicitly providing a valid location for the `store` parameter. When no storage is explicitly specified by the user (by passing a `0` value), automatic memory management is assumed and the system allocates dynamic storage for the replication data. It keeps track of the dynamic storage across process replications.

- Constructs to specify computation state.

  A predefined structure `BspSystemState` can be used to declare variables that hold the specific or common system state. The function `bsp_init_system_state` can be used to initialize a `BspSystemState` variable. Using the function `bsp_set_system_state`, the state variable can be made to hold variables that comprise the computation state (specific or common system state). The specific and common system states can be specified for a computation superstep using the constructs `bsp_specific_system_state` and `bsp_common_system_state`.

- Constructs to specify a computation superstep.

  The constructs `bsp_comp_sstep` and `bsp_comp_sstep_end` are used to delimit a computation superstep. The replication and recovery mechanism is embedded into these constructs; the process of data replication, detection of failures and recovery is transparent to the user.

- Recovery Function.

  The predefined function `RecoveryFunction` is executed after restoring the computation state of a failed process from the backup. The user must supply the code required for any operations required for recovering the computation state of a failed process. Specification of the recovery function is optional.

Figures 6.2 - 6.5 illustrate the use of BSP constructs for adaptive parallelism. These examples were taken from a C++ implementation of a plasma simulation using the adaptive replication system.

Figure 6.2 shows the constructs provided by the user layer. These constructs are described above.

Figure 6.3 illustrates the use of these constructs. Case (a) illustrates the specification of a variable `plasma_region` as the replication data for which static storage is available in `plasma_region_backup`. The construct allows the user to supply a string `‘‘plasma_region’’`, which can later be used to refer to this replication data (refer to Figure 6.4). In case (b), the replication data consists of a vector of objects of a user defined type `ChargedParticle`. No static storage is available for this replication data (specified by a `0` for storage). The tag for the replication data is `‘‘PLASMA_POS’’`. In case (c), the replication data consists of each of the `SYSLEN_MX` arrays, each a 2 dimensional array with no static storage available for replication data. All the replication variables use the same tag `FORCE_FIELD_X`, however an index variable `<i>` is used to distinguish between the `SYSLEN_MX` replication variables.

Figure 6.4 illustrates the use of the constructs to specify the computation state of a component process. The construct `BspSystemState` is used to create a variable to hold the computation state. The variable is initialized with the construct `bsp_init_system_state`. The construct `bsp_set_system_state` is used to include each of the replication data defined in Figure 6.3 in the computation state. Note the use of tags to refer to the replication data.

Figure 6.5 illustrates the use of the extended BSP construct for the computation superstep. The specific and local system states must be specified for each computation superstep. The computation superstep requires no additional constructs; adaptive replication and recovery of failed computations are done transparently.

```
/* Constructs to specify a computation superstep */
bsp_comp_sstep(stepid);
bsp_comp_sstep_end(stepid);

/* Constructs to specify replication data and allocate storage */
bsp_replication_data(void* data, long nbytes, void* store, char* tag,
                              int subscript);
bsp_setup_replication_environment();

/* Constructs to specify Computation State */
struct BspSystemState;
bsp_init_system_state(BspSystemState* bss);
bsp_reset_system_state(BspSystemState* bss);
bsp_set_system_state(BspSystemState* bss, char* tag, int subscript);
bsp_specific_system_state(BspSystemState* bss);
bsp_common_system_state(BspSystemState* bss);

void RecoveryFunction();
```

Figure 6.2: Adaptive parallel extensions to the Oxford BSP Library (User Layer)

6.3 Implementation of the Adaptive Replication System (Replication Layer)

We have implemented the adaptive replication system as additional layers on top of the Oxford BSP library. The ARS is available as a library of C functions and can be used by parallel applications in the same way a BSP library is used. In implementing the prototype, we have assumed a replication level of one. That is, a process can act as a backup for its immediate predecessor only. The prototype is implemented on Sun Sparcstations using the Solaris (SunOS 5.5) operating system.

```
/* case (a): (static) storage available for replication data */
bsp_replication_data((void*) &plasma_region, sizeof(plasma_region),
                     (void*) &plasma_region_backup,
                     "PLASMA_REGION", -1);

/* case (b): storage to be allocated by the BSP library */
bsp_replication_data((void*) elec_pos, PTMAXNP * sizeof(ChargedParticle),
                     0,"PLASMA_POS", -1);

/* case (c): 2 dimensional array, with no static storage for replication data */
for(i=0; i < SYSLEN_MX; i++)
  bsp_replication_data((void*) ForceFieldX[i], SYSLEN_Y*sizeof(Scalar),
                       0,"FORCE_FIELD_X", i);
```

Figure 6.3: Use of extended-BSP constructs to specify replication data

```
BspSystemState plasmaState = new BspSystemState;
bsp_init_system_state( plasmaState );
/* Specify the data for the state variable, using symbolic names */
bsp_set_system_state(specific, "PLASMA_REGION", -1);
bsp_set_system_state(specific, "PLASMA_POS", -1);
for(i=0; i < SYSLEN_MX; i++)
   bsp_set_system_state(specific, "FORCE_FIELD_X", i);
```

Figure 6.4: Use of extended-BSP constructs to specify computation state

It makes use of the checkpoint based migration scheme of Condor [10] for process migration. It should be noted that our protocol for adaptive replication scheme can be applied to other message passing libraries such as MPI [50]. The only requirement is that the application be written in the BSP-style, using a sequence of computation and communication supersteps.

The replication layer forms the core of the adaptive replication system. It is built on top of the extended BSP library and implements all the functionality of the adaptive replication system listed in Section 6.2.3. In this section we describe the implementation of some of the important functions performed by this layer.

6.3.1 Failure Detection and Replication of Computations

In the adaptive replication scheme, a process starts replicating for its predecessor when it concludes that its predecessor has failed. Failure detection is based on

```
bsp_specific_system_state( plasmaState );
bsp_local_system_state( localCharge );

bsp_comp_sstep( bsp_step );
CalcEField( vpm, energy );
InitChargeDensity();
energy.ke( 0.0 );
Advance( elec_pos, elec_vel );
bsp_comp_sstep_end( bsp_step );
```

Figure 6.5: A BSP computation superstep with adaptive replication.

the receipt of a message from the predecessor indicating successful completion. Not receiving the message is interpreted as the failure of the predecessor. Failure detection is a tricky issue in distributed system design as there is no way to distinguish between a failed process and a process that is simply slow. In a heterogeneous network the computations on individual workstations often proceed at different speeds owing to differences in processor speed, characteristics of work load on the individual machines, etc. To compensate for the differences in processing speed, a *grace period* can be used to allow a slow predecessor to complete its computations before concluding that the predecessor has failed. The grace period can be specified by the user in an include file. However, using a grace period also delays replicating for the predecessor when required. Our implementation allows the user to specify the grace period. However, based on experimental results, we have not used a grace period with the applications we tested. A process starts replicating for its predecessor if it has not received a message of successful completion from the predecessor by the time it finishes its own computations. However, to avoid unnecessary migrations, we abort the replicated process and allow the predecessor to continue if the predecessor completes its computations before the replicated process or if it completes before the synchronization is achieved. This results in a nice property of the adaptive replication scheme - any processor that is twice as slow as its successor and slower than all other processes participating in the parallel computation is automatically dropped from the parallel computation and a new available host is chosen in its place. This allows the application to choose faster machines for execution from the available machines.

6.3.2 Process Migration

When the replicated process finishes its computation and successfully completes synchronization on behalf of the failed predecessor, it assumes the identity of the predecessor and can continue in the parallel computation as a legitimate member. The computation can continue albeit at a reduced degree of parallelism, as there are now two processes on the same host. However we would like to maintain the parallelism by migrating the process to a new host if one is available. Migration is achieved by checkpointing the process and restarting the process from the checkpoint on the new host. Checkpointing and restarting are done using Condor [44].

6.3.3 Coordination of Distributed events

The adaptive replication system implements a distributed fault-tolerance system. The ARS needs to coordinate and control various events that occur at the component processes. Coordination and control of the events is handled by a combination of signals, messages and locking. The following is a list of events that are handled by ARS.

- Completion of the computation by a component process is communicated to the successor by a message indicating successful completion. Not receiving the message is interpreted as the failure of the predecessor.

- Migration of a process to another available host is achieved by checkpointing the process and restarting the process from the checkpoint on the target host. Restarting the process cannot start before the checkpointing is complete. Coordination of these events is handled through file locking.

- A process delayed due to a transient failure of its host and whose computations have been successfully replicated needs to be terminated. Termination of a lagging process is done by sending it a `SHUTDOWN` message. A process terminates itself upon receiving a `SHUTDOWN` message.

- A newly created process replicating a delayed process needs to be terminated if the delayed process manages to finish its computations before the new process.

Termination of the replicated process is done by the parent process which created the new process.

6.4 Host Monitor

The status of the host machines is monitored by the host monitor. A monitor daemon exists on each of the host machines. The monitor periodically collects information on the usage of the machine such as keyboard and mouse activity of the console user and load on the machine. This information is obtained from the kernel structures for accounting information. The monitor determines the host status based on these data using the algorithm presented in the next section.

6.4.1 Algorithm to Determine Host Status

The host status is determined based on the activity of the console user as well as the load on the host machine. Figure 6.6 shows the algorithm (written in pseudo code) used to determine the host status. We will explain the algorithm briefly here.

The console is considered to be idle if there is no user on the console or if the console user is idle for at least a specified period of time (`MIN_IDLE_PERIOD`). In addition to the console activity, the load average within the last one minute is also considered to determine the host status. Different load thresholds are used for idle and active consoles, with the load threshold when the console is active being less than the corresponding load threshold when the console is idle. Thus the parameter `LOAD_THRESHOLD_FOR_ACTIVE_CONSOLE` has a value that is less than the value of `LOAD_THRESHOLD_FOR_IDLE_CONSOLE`. The number of consecutive periods in which the average load is less than the threshold load values is maintained (`low_load_period_count`). If the load average is more than the predetermined threshold value, then the host status is determined to be unavailable. However, if the load average is less than the threshold load value, then the host status is determined based on the number of consecutive periods in which the load average is less than the threshold. If this count is less than a specified number (`MIN_COUNT`), then the host status is determined to be *unavailable*. In all other cases the host status is determined to be *available*.

Since the remote process imposes additional load on the host machine, a lower load threshold is used for new processes that want to enter the host machine than for remote processes that are already executing on the host. This avoids the host machine becoming unavailable as soon as a remote process is allowed to execute on the host.

6.5 Summary

In this chapter we presented the protocol used by the participating processes to detect failures and replicate computations of failed processes. We presented the design of the adaptive replication scheme within the framework of the BSP model. We also described the implementation of a prototype system. We extended the Oxford BSP library with additional features necessary for the implementation of the adaptive replication scheme. The adaptive replication scheme was built on top of the extended BSP library in layers. The layered approach helps to insulate the application programs from changes to the implementation of the adaptive replication scheme or the replication protocol. By implementing the replication layer for other architectures, we can maintain the portability of applications using our library.

```
Find the console idle time.
//If (console idle time < MIN_IDLE_PERIOD)
//Then console is not idle
//Else console is Idle

Find the machine load average.
If load_average < LOAD_THRESHOLD_FOR_ACTIVE_CONSOLE
Then
  low_load_period_count++;
Else
  low_load_period_count = 0;

If the console is idle,
Then
  if the load average > LOAD_THRESHOLD_FOR_IDLE_CONSOLE
  Then
    status = UNAVAILABLE;
  Else
    status = AVAILABLE;
Else //Console is busy
  if the load average > LOAD_THRESHOLD_FOR_ACTIVE_CONSOLE
  Then
    status = UNAVAILABLE;
  Else
    If current status == AVAILABLE
    Then
      status = AVAILABLE;
    Else
      low_load_period_count = number of consecutive periods during which
        load is less than LOAD_THRESHOLD_FOR_ACTIVE_CONSOLE.
      if (low_load_period_count < MIN_COUNT)
      Then
        status = UNAVAILABLE;
      Else
        status = AVAILABLE;
```

Figure 6.6: Algorithm to determine the host status

CHAPTER 7
APPLICATION OF ADAPTIVE REPLICATION SCHEME TO PARALLEL COMPUTATIONS

In chapter 4, we analyzed the performance of the adaptive replication scheme and categorized parallel applications based on the cost of data replication as *computation dominant* and *replication dominant* parallel computations. To recapitulate, computation dominant parallel applications are those in which data replication can be done without a significant overhead and hence the cost of a computation step is determined by the cost of computation alone. Replication dominant applications are those in which the cost of data replication adds a significant overhead to the cost of the computation step. In this chapter we consider the application of the adaptive replication scheme to parallel computations. We consider two applications that illustrate the performance of the scheme for computation dominant applications and data replication dominant applications.

The network of workstations used for these experiments consists of Sun Sparcstations connected by a 10Mbps Ethernet, which can be considered to be slow compared to the networks available in the market today. Hence, the performance of parallel applications using our network will be less than the performance which can be obtained by a faster network. The slow network also affects the performance of the adaptive replication scheme by increasing the cost of data replication and migration.

7.1 Plasma Simulation

Section 3.2.1 gives an overview of plasma simulation. Here we describe aspects of plasma simulation that are relevant to the application of adaptive replication scheme. The operations in the computation step modify the position and velocities of the particles and the charge distribution on the grid. Hence, the computation state data that need to be replicated include the positions and velocities of the particles and the grid charge. However, at the beginning of each superstep, all processors have the same global charge distribution and hence the charge data need not be replicated on a remote host. Using the terminology introduced in Section 6.1, the positions and velocities of the particles constitute the Specific System State and the portion of the grid charge on each process constitutes Common System State. Since the grid charge is modified by the computation step, each process can save this data locally which it can use to restore a failed predecessor. Checkpointing data locally when possible reduces the amount of data communicated for data replication. Due to the overhead associated with the communication of computation state, this application can be categorized as a replication dominant application (also see discussion in Section 7.3). In plasma simulation, the computation state data cannot be recomputed and hence the Recovery Function is empty.

7.2 Maximum Independent Set

The graph search algorithm used to find the maximum independent set of a graph is described in Section 3.6. Here we will describe the data to be replicated under the adaptive replication scheme. The adjacency matrix of the original graph is replicated on each processor and constitutes the common system state. Since this data is not modified by the computation step, no local checkpointing of this data is necessary. During local search of a subgraph, each of the processes search a different subgraph. Data about the subgraph that is being searched locally at each process constitutes the specific system state at that process. However, the virtual subgraph used in local search can be recreated from the connectivity information, the level of recursive search and the identity of the failed process. Hence recovery of a failed process can be achieved without replicating any computation state data. This application thus illustrates the performance of a computation dominant application.

7.3 Results

The results presented in this section with the two applications described above have been obtained with simulated transient processors. (Results on a real network are presented in Section 7.5.) A timer process maintains the state of the transient processor - *available* or *unavailable*. The duration of the available and nonavailable

Single Proc.	Degree of	Dedicated Processors	Transient Processors with Adaptive Replication			Trans Procs without ARS
Exec Time	Para-llelism	Exec Time	Mean Time	Min (#Moves)	Max (#Moves)	Mean Time
10300	3	3650	4350	3950 (2)	4790 (8)	12400
	6	1840	2340	1990 (1)	2900 (10)	12900
	12	980	1620	1000 (0)	2700 (9)	26650

Table 7.1: Execution times of maximum independent set problem on dedicated processors, on transient processors with adaptive replication and on transient processors without adaptive replication. For the runs on transient processors with adaptive replication, number of migrations during the lifetime of the parallel computation (#moves) is listed in parentheses. All times shown are wall-clock times in seconds.

periods are determined using a random function according to a uniform exponential distribution. State changes from an available to a nonavailable state and vice versa are conveyed to the application processes on the host machine via signals.

Table 7.1 shows the execution times of maximum independent set problem on transient processors using the adaptive replication scheme with $t_a = 40$ minutes and $t_n = 20$ minutes respectively. These values for t_a and t_n are within the range of values reported in earlier works [52]. The three columns represent the mean, minimum and maximum execution times of a number of trials. The measurements were taken on a network of Sun Sparc 5 workstations connected by a 10 Mbps Ethernet. The degree of parallelism used in the simulations is much smaller than the number of processors and therefore, migration to an available processor was always possible.

The execution times of the runs on transient processors using the adaptive replication scheme were compared with the execution time on dedicated processors and with execution time on transient processors without using the scheme. The execution time on a single processor is also shown for reference. As can be seen from these timings, the runs on transient processors using the adaptive replication scheme compare favorably with the runs on dedicated processors. Figure 7.1 shows a plot of these timings.

Our measurements indicate that a significant amount of computation was per-

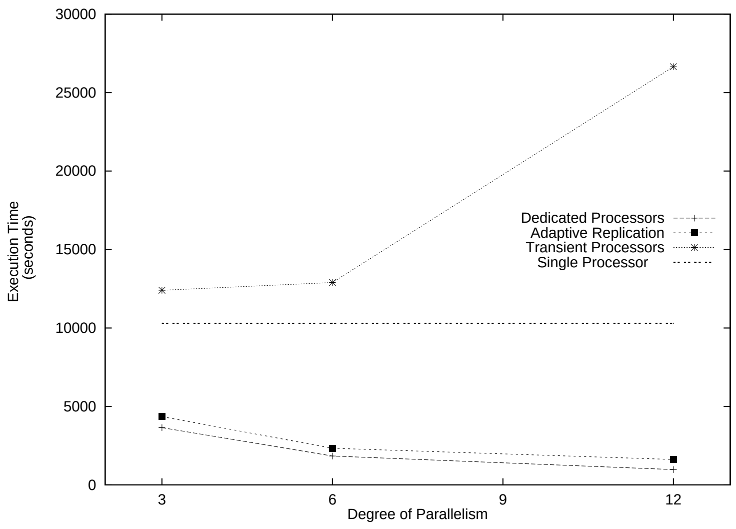

Figure 7.1: Plot showing execution times of maximum independent set problem on transient processors using adaptive replication, on transient processors without adaptive replication and on dedicated processors. Execution time on a single processor is shown for comparison purposes.

formed using idle workstations. As mentioned in 5.1.2, parallel runs using the adaptive replication scheme use the user's own host for one of the processes and use the idle machines in the network for the remaining processes so that when using a parallelism of p, a fraction of $\frac{p-1}{p}$ of the total computation is performed by the idle machines. In this case, a significant proportion of work - for example, 84% when using 6 processors and about 92% when using 12 processors, was done using the idle processors.

For the runs on dedicated processors, parallel efficiency is given by $\frac{T_s}{pT_p}$, where T_s is the sequential execution time, T_p is the parallel execution time and p is the number of processors. For the runs on nondedicated processors, p is replaced by the effective number of processors, $p_{\mathit{eff}} = 1 + (p-1)\frac{t_a}{t_a+t_n}$, since each processor is available only for a fraction of $\frac{t_a}{t_a+t_n}$. For the values of t_a and t_n used for these runs, $p_{\mathit{eff}} = \frac{2p+1}{3}$. For the dedicated runs, parallel efficiency ranges from nearly 100% (for 3 processors) to 88% (for 12 processors). For the nondedicated runs using adaptive replication, these values range from nearly 100% (for 3 processors) to 76% at 12 processors. The corresponding values for nondedicated runs without adaptive replication are 36% and 5%. Thus the adaptive runs are nearly as efficient as the dedicated runs and much more efficient than the transient processor runs.

Table 7.2 shows the results of application of the adaptive replication scheme to plasma simulation with $N = 3,500,000$ particles. Figure 7.2 shows a plot of execution time on transient processors with and without adaptive replication for degrees of parallelism of 4, 8 and 12. These measurements were obtained using $t_a = 30$ minutes and $t_n = 20$ minutes respectively. As mentioned in Section 7.1, due to the overhead associated with the communication of computation state in each step, simulation runs on transient processors using the adaptive replication scheme take longer to execute compared to the runs on dedicated processors. However, even in this case, adaptive replication scheme is relevant for the following reasons. The execution time on transient processors with adaptive replication is still much smaller than the execution time without adaptive replication. Since the duration of the simulation is less than the mean available period, number of transient failures is very small and hence their impact on program execution is not significant at small degrees of parallelism. However, the impact of transient failures grows rapidly with the number of processors, as is evident from the plot. Parallel efficiency of adaptive runs as defined above is much higher compared to the runs on transient processors, especially at larger number of processors. The parallel efficiency of adaptive runs, as defined above, ranges from 34% on 4 processors to 16% on 12 processors, whereas the corresponding figures range from 35% on 4 processors to about 2% on 12 processors for runs on transient processors without adaptive replication. (We could not run the simulation on a single processor due to insufficient memory. The values of parallel efficiency mentioned above are based on a sequential execution time of 3360 seconds, which is estimated from the execution time of a parallel run on 4 dedicated processors assuming 100% efficiency.) As explained above, simulation runs using transient processors with the adaptive replication scheme use idle machines for a significant fraction of their work. Further, the simulation used for our measurements was too large to fit on a single workstation and hence single processor runs were not even possible. For simulations that are too large to fit on a single workstation, parallel runs are mandatory. When dedicated machines are not available for parallel computation, adaptive replication scheme ensures that parallel runs using

Single Proc.	Degree of Para-llelism	Dedicated Processors	Transient Processors with Adaptive Replication			Trans Procs without ARS
Exec Time		Exec Time	Mean Time	Min (#Moves)	Max (#Moves)	Mean Time
not possi-ble	4	840	3500	3340 (3)	3774 (6)	3400
	8	750	3100	2583 (5)	3503 (11)	20300
	12	620	2700	2150 (5)	3080 (20)	26500

Table 7.2: Execution times of plasma simulation on dedicated processors, on transient processors with the adaptive replication and on transient processors without adaptive replication. For the runs on transient processors with adaptive replication, number of migrations during the lifetime of the parallel computation (#moves) is listed in parentheses. All times shown are wall-clock times in seconds.

idle workstations complete in a reasonable time.

Any approach intended to tolerate transient failures will necessarily incur some overhead to checkpoint the computation state of the processes. Overhead incurred by replication of computation state as done in the adaptive replication scheme (which can be considered a form of diskless checkpointing) is no larger than the overhead caused by checkpointing to disk. The network used to obtain the measurements is a 10 Mbps Ethernet, which is quickly becoming obsolete. With a faster network such as an ATM network or a fast Ethernet, the overhead due to data replication should be much smaller.

7.4 Improved Parallel Graph Search Algorithm

In the graph search algorithm described in Section 7.2, the connectivity information (the adjacency matrix) is replicated on all the processors. Replication of the adjacency information improves the efficiency of the parallel graph search by reducing the amount of data communicated. Replication of the adjacency information also reduces the state information that is required for recovering from transient failures. However, replication of the adjacency matrix on all participating processors limits the maximum size of the problem that can be solved. It is desirable to find a parallel graph search algorithm that allows for solution of problems of size

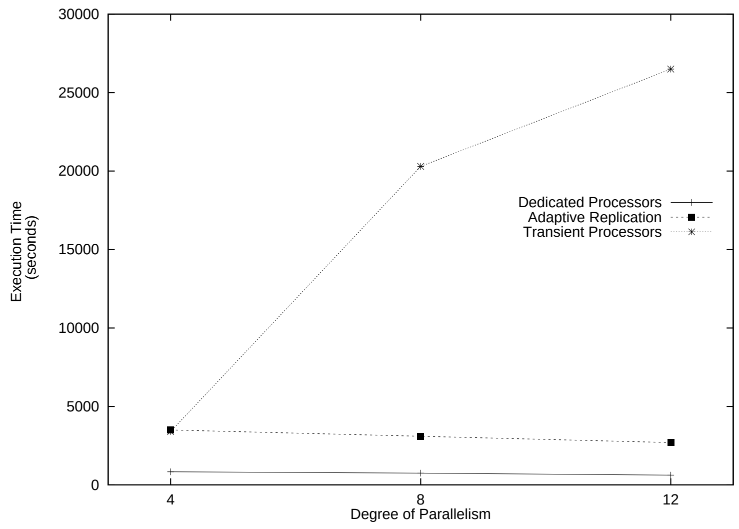

Figure 7.2: Plot showing execution times of plasma simulation on transient processors using adaptive replication, on transient processors without adaptive replication and on dedicated processors.

larger than that can be solved on a single processor. In this section, we describe an improved graph search algorithm with these characteristics.

7.4.1 Improved Parallel Graph Search Algorithm

In the improved graph search algorithm, the adjacency matrix is partitioned among the participating processes in the following manner. The rows of the adjacency matrix are partitioned among the participating processes such that each process contains the complete connectivity information for a subset of the vertices. We refer to this subset of vertices as *belonging* to the corresponding processor. As explained in Section 3.6, at each level of the recursive depth first search, a new vertex is added to the independent set being constructed and all vertices in the current graph that are adjacent to this vertex are deleted to form the vertex list for the new graph to be searched. Since each processor contains only a portion of the adjacency matrix, each processor needs to obtain adjacency information for vertices that *belong* to other processors. As explained in Section 7.2, when the subgraphs generated are of sufficient granularity they are searched locally on one of the participating processors. To avoid communication during the local search, each participating process needs to have adjacency information for all pairs of vertices in its subgraph. For this purpose, before starting local search on its subgraph, each processor obtains adjacency information for vertices in the subgraph that belong to other processes. Once the adjacency matrix is constructed for the subgraph to be searched, the processors do not need to communicate during the local search.

7.4.2 Performance of the Improved Parallel Graph Search Algorithm

Consider a graph with n vertices and a probability of d that any two vertices are connected. The adjacency matrix is partitioned among the processors equally and hence each processor contains the complete adjacency information of n/p vertices or n^2/p entries, where p is the number of processors. The adjacency matrix is represented as bit patterns. If W is the word size on the host machine, then the amount of storage required for the local partition of the adjacency matrix is n^2/pW. At each level of the recursive search, a vertex with minimum degree is added to the independent set and all vertices that are adjacent to it are deleted to form the virtual graph at the next level. Thus the number of vertices of the virtual graph to be searched decreases exponentially at a rate of $(1-d)$. The communication required at each level of the recursive search contains the adjacency information for each of the vertices that belong to other processors. At a given level of the recursive search l, the number of vertices of the subgraph is approximately $n(1-d)^l$. Since the adjacency matrix is partitioned equally among the processors, roughly a fraction of $\frac{p-1}{p}$ of these vertices belong to other processors. With bit representation for the rows of the adjacency matrix, the amount of data required at each processor is approximately $\frac{p-1}{p}n(1-d)^l\frac{n}{W}$. The amount of communication decreases exponentially with the level of the search tree. Finally each processor needs to construct a complete adjacency matrix for the subgraph to be searched locally. If m is the number of vertices of the subgraph to be searched locally, then the amount of adjacency data required at each processor is $\frac{p-1}{p}m\frac{m}{W}$. The number of distinct subgraphs of a graph with m vertices is 2^m, so the cost of searching all the subgraphs is exponential in the number of vertices of the subgraph m and the communication can be performed without losing parallel efficiency.

7.4.3 Adaptive Behavior of the Improved Parallel Graph Search Algorithm

The adaptive performance of the graph search algorithm depends on the amount of data to be replicated. Since the adjacency matrix is partitioned among the processes, the adjacency matrix partition of each process needs to be copied to a backup process. The replication of adjacency matrix partition is used to recover the predecessor in case of a failure. The adjacency matrix partitions can be replicated at the beginning of the search because these need not be updated in each computation step. However, when a process performs recovery for its failed predecessor and assumes its identity, it needs not only the failed process's adjacency matrix partition but also a replica of the adjacency matrix partition of the failed process's predecessor. The new process needs this replica only in case of a failure of its predecessor. Therefore, updating the adjacency information of the predecessor needs to be done once per recovery. The cost of refetching the adjacency matrix partition can therefore be included in the cost of recovering a failed process. Figure 7.3 illustrates recovery of a failed process (2) by its successor process (3). Process 3 creates a child process which performs recovery for process 2 and assumes its identity for subsequent computations. Since the new process still contains the replica of adjacency matrix partition of process 2, this replica needs to be updated with the adjacency matrix partition of process 1 (its predecessor) so that it can recover process 1 in case of a failure.

In addition to the adjacency matrix partition, the adjacency matrix of the subgraph that is searched locally also needs to be replicated. However, since the subgraphs searched locally on the participating processors differ only slightly, we can avoid replicating the adjacency matrix of the local subgraphs if we construct the adjacency matrix for the largest of these subgraphs. This is the approach followed in our implementation. Since the subgraphs differ slightly from each other, a mapping that identifies these vertices needs to be replicated on the backup process. This cost is proportional to the number of vertices in the original graph. Thus the amount of communication required for replication during normal execution is smaller than the communication inherent to the algorithm.

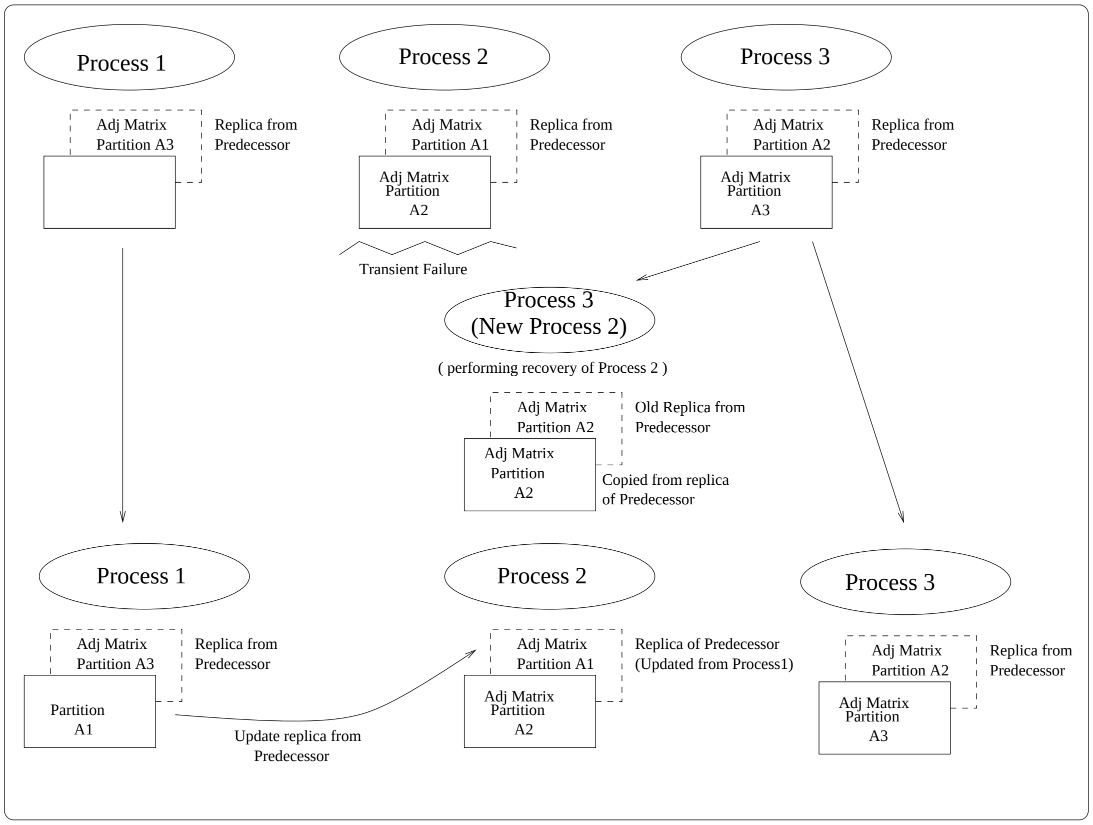

Figure 7.3: Illustration of recovery of a failed process in the improved graph search algorithm

7.4.3.1 Performance Characteristics of Improved Graph Search

The execution times of the improved graph search are plotted in figure 7.4. The execution times for a 5000 vertex graph with the probability of edge connectivity of 0.54. As can be seen from the plots, the algorithm obtains good speedups on dedicated processors.

7.5 Application of Adaptive Replication to Real Networks

As explained in Section 7.3, the results presented above have been obtained with simulated transient processors. To demonstrate the applicability of the Adaptive Replication System to a real situation, we tested the system in a real environment on the Computer Science Department Network at Rensselaer Polytechnic Institute. The improved graph search algorithm described in Section 7.4.1 is run on the workstations in the Computer Science department. The host monitor described

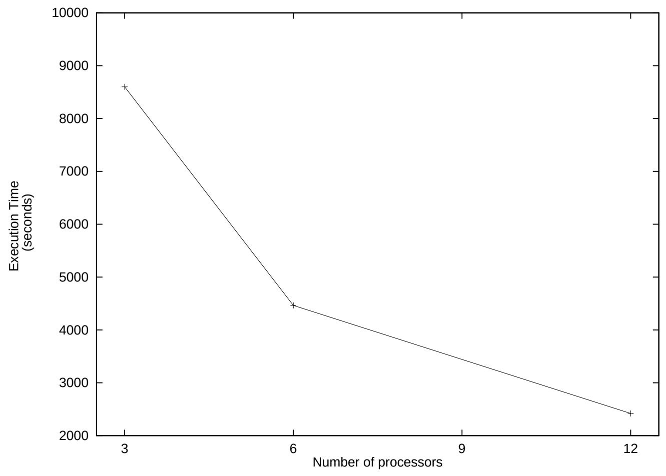

Figure 7.4: Speedup characteristics of improved graph search algorithm

in Section 6.4.1 running on the workstations determines the status of the host. The processor pool used for these runs consisted of about 20 machines that included Sparc 5 (Models 110 and 70) and Sparc 20 processors. The runs used a degree of parallelism of 6, with 5 of them using nondedicated machines. To compensate for the difference in speeds of the different machines, a grace period of 60 seconds is used. That is, a process will wait for 60 seconds after it finishes its computation to replicate for a failed process. For 10,000 vertices with a mean probability of connectivity of 0.54, the execution time on nondedicated processors is about 12 hours compared to about 10.5 hours on dedicated processors.

The maximum size problem that can be solved on a single processor is a graph of 10,000 vertices. Using the improved graph search algorithm, we are able to solve graphs of size 15,000 vertices. The execution time for a graph with 15,000 vertices and a probability of connectivity of 0.56 is about 7.5 hours when using a degree of parallelism of 12. The corresponding execution time on nondedicated processors is about 10 hours. Parallel run on dedicated processors used the faster processors (Model 110) while the run on nondedicated processors used a mixture of fast and slow processors (both Model 110 and Model 70), so part of the execution is on the slower processors. Scalable parallel algorithms are essential to solve problems that are too large to fit in the memory of a single processor. Adaptive replication scheme allows efficient execution of parallel runs on nondedicated processors.

7.6 Summary

In this chapter we presented the application of adaptive replication scheme to two parallel computations - a computation dominant and a replication dominant application on simulated transient processors. The results demonstrate that we can execute computation dominant applications efficiently on nondedicated workstations. We also demonstrated the capabilities of the adaptive replication scheme by executing the graph search program on idle workstations on the network of Sun workstations in the Department of Computer Science at Rensselaer Polytechnic Institute.

CHAPTER 8
CONCLUSIONS AND FUTURE WORK

In the preceding chapters we demonstrated applicability of BSP based parallel processing to networks of workstations with applications from plasma simulation, graph search and finite element modeling. We proposed an approach to adaptive parallelism in a network of transient processors within the framework of the BSP model. We presented an analysis of our scheme, design of the Adaptive Replication System and application of the system to specific parallel computing applications. We shall now examine this work in a broader perspective - including its scope, applicability, contribution, limitations, extensions and future work.

8.1 Discussion

The BSP model allows the programmer to gain insight into the structure of the parallel computation and to use the insight to improve the performance by changing the data distribution. We have demonstrated this ability of the BSP model by implementing the plasma simulation in the BSP model. We used BSP analysis techniques to improve the performance on a network of workstations.

The BSP model is intended by its author to be a general purpose programming model. Since its introduction, the model has been used to implement a variety of parallel algorithms (see section 1.3), and programming environments are being developed. In addition to plasma simulation, we have used the model to implement parallel algorithms in the areas of graph search, finite element computation and other simulations.

Despite the obvious advantages of the BSP model as a basis for predictable parallel computing, BSP has not become popular outside the academic and research world. However the advantages of the BSP model go beyond simple parallel programming. The barrier synchronization in the BSP model ensures that all participating processes reach a globally consistent state. Although much other research has been done in this area, BSP makes the implementation of consistent checkpoints much simpler. We have used this property to devise a simple approach to tolerate transient failures of host processors.

Our approach to adaptive parallelism is based on replicating the computations of a failed process on another participating process. Thus the recovery of a failed process is done on an available (idle) workstation. Replication of computations of the failed process is made possible by eagerly replicating the computation state of each process on a designated peer process which acts as a backup. Thus the cost of tolerating transient failures depends on the relative cost of replicating the computation state compared to local computation. Intuitively, algorithms in which the local computation is larger than the cost of communicating the state to another process are suitable candidates for application of adaptive replication scheme. Therefore our approach is suitable to computation intensive parallel applications.

Even though our approach to adaptive parallelism is designed within the framework of the BSP model, it applies to all parallel applications written in BSP-style as a sequence of computation and communication steps. Keeping this in view, the ARS library is designed in layers. The replication layer is built on top of the native communication library and helps to insulate the user layer from implementation details. The parallel applications are written using only the constructs in the user layer and the application programmer is insulated from changes to lower level layers. Thus the adaptive replication library can be reimplemented using other communication libraries such as MPI as long as those communication libraries support constructs for BSP-style parallel programming.

8.2 Contributions

This work makes contributions to our objectives stated in Chapter 1 to achieve BSP-based parallel computing and to enable parallel computations adapt to the computing environment. The thesis makes the following contributions to the theory and practice of bulk-synchronous and adaptive parallel computations:

- Demonstrated the applicability of the BSP model to a variety of problems. We demonstrate how the BSP model can be used to analyze and tune the application to improve its performance on a given parallel architecture. We used BSP analysis techniques to gain insight into the computation and used the knowledge to change the data distribution to allow the simulation to run efficiently on a network of workstations.

Demonstrated the versatility of the BSP model by applying it to problems in plasma simulation, graph search and finite element modeling.

- Analyzed the impact of transient failures on the execution time of frequently synchronizing parallel computations for exponentially distributed available and nonavailable periods.

- Developed a general approach to tolerate transient processor failures based on eager replication of computation state and lazy replication of computations within the framework of the BSP model.

- Analyzed the performance of a parallel application using the Adaptive Replication Scheme on a network of transient processors. The analysis proves that, for scalable parallel application under suitable conditions, the adaptive replication scheme provides scalable speedups.

- Designed a protocol for the replication of computation state and replication of computations for a replication level of one. The protocol can be generalized to higher levels of replication.

- Extended the Oxford BSP model to include dynamic process management and virtual synchronization, allowing it to be used as a basis for a virtual BSP Computer.

- Designed a runtime system to support adaptive replication scheme within the BSP model. The design incorporates a layered approach to provide for portability and extensibility.

- Implemented a prototype of the adaptive replication system based on the extended BSP library. The prototype implements a distributed fault-tolerant system and uses TCP/IP (socket) based message passing, signals, signal handlers, process management and locking to coordinate distributed events and to avoid deadlocks.

- Demonstrated the adaptive capability of the system by applying it to two different applications and measuring their performance for simulated transient processors.

- Developed monitoring software to monitor the status of a host machine taking into account activity of the console user including keyboard and mouse activity and cpu load. The criterion for determining the status of the machine can easily be configured by the user.

- Demonstrated the adaptive capabilities of the system by running the application on the Computer Science Network at the Rensselaer Polytechnic Institute.

8.3 Future Work

Future work needs to focus on the following areas. Improvements to the communication network and communication protocols increase the performance of parallel applications on dedicated as well as nondedicated networks. Another important objective is portability across heterogeneous processor architectures. Extensions to the scheme to incorporate multiple virtual processors on the same host will make the scheme more general and will improve its ability to balance the load according to the capacity of the host machines. Characterizing and predicting host status will help in anticipating failures and avoiding host machines that are prone to failure. In addition to the above, specific enhancements can be made to the system to improve performance in some cases.

The adaptive replication scheme seeks to reduce the overhead of data replication by overlapping local computation with the communication associated with data replication. Overlapping the computation and communication requires that the communication be done in a separate thread within the user process. Thus effective overlapping of computation and communication may require the use of a multi-threaded communication library.

Communication over TCP sockets incurs a significant overhead as a result of the large number of layers in the TCP/IP protocol stack. A number of researchers are looking into reducing this overhead through the use of new protocols and through new implementations of the TCP/IP protocol. Communication libraries with low overhead will reduce the cost of communication and replication.

Another area that is worth attention is portability of the scheme across heterogeneous processor architectures. There are two constraints in seeking this goal. First, to enable data replication across processor architectures, routines must be supplied to marshal and unmarshal replication data from the data format of the sender to that of the receiving processor. Second, we need a portable scheme to support migration of processes across architectures. The use of the Java programming language plus CORBA for interoperability could be explored to address the issue of process migration across heterogeneous architectures.

The adaptive replication system operates by treating the participating processors as virtual processors and continually mapping the set of virtual processors to available host processors. In the current design ARS assigns each virtual processor to one of the available host machines such that no two virtual processors are assigned to the same host. The virtual processor-to-host machine mapping is based on the availability of the host machine. Currently the *availability* of a workstation is treated as discrete. A host machine is either busy and therefore not available or free and therefore available for the parallel computation. However, as the processing power of workstations increases, it may be possible to view machines in terms of their *degree of availability.* With such a concept, workstations will be able to contribute cpu cycles to the parallel computation in proportion to the percentage of idle time at any given time. To take advantage of the cpu cycles that can be offered by a given a host processor, the protocol should be extended to support multiple virtual processors on a single workstation.

The adaptive replication system enables the parallel computation to tolerate and make progress in spite of transient failures of host machines. Even though the scheme can tolerate real failures of host machines in certain situations, it is desirable to extend the scheme or supplement it with features to provide tolerance from real host failures. However, since adaptive parallelism is primarily a means of improving performance in a nondedicated environment, care should be taken to avoid any adverse impact on the performance.

The ability to adapt to a computing environment can be significantly improved if we can predict the occurrence and the duration of transient failures. For this, we need a better understanding of the host load characteristics and distribution of the available and nonavailable periods. Sophisticated mathematical models may be needed to predict the occurrence of and the duration of transient failures.

8.3.1 Enhancements to the Adaptive Replication System

A few enhancements to the adaptive replication system can help to improve performance for some applications.

8.3.1.1 Incremental Replication

Applications in which changes to the computation state are small compared to the size of the computation state can reduce the cost of data replication by replicating only the changes to the computation state. In the modified scheme the computation state of a component process is replicated in full once. Subsequently, only changes to the computation state since the last replication are sent to the backup processes. The process receiving the replicated data applies the changes to its backup copy to get an updated copy of the computation state. However, when a process successfully replicates the computations of one of its predecessors and replaces the predecessor in the parallel computation, the new process must again receive a full copy of the computation state of its predecessors for which it is acting as a backup. Therefore this enhancement is useful when the transient failures are not too frequent.

8.3.1.2 Data Compression to Reduce Communication Cost

The cost of data replication can be reduced in some cases by compressing the data to be communicated. The receiving process then needs to decode the compressed data to get the actual data, but only in case of failure of the sender.

Compression of the data to be replicated is useful only when the decrease in communication cost justifies the cost of data compression and decompression at the sending and receiving processes.

8.3.1.3 Other Enhancements

Process migration time can be improved by writing the checkpoint to the local disk on the target machine rather than to the network file system. Restarting the process from the checkpoint on the target machine can now be done off the local disk, without involving the communication network.

BIBLIOGRAPHY

[1] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[2] R. Alonso and L. L. Cova. Sharing Jobs Among Independently Owned Processors. In *Proc. 8th Intl. Conf. Distributed Computing Systems*, pages 282–288, San Jose, California, June 1988.

[3] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15:54–64, Feb 1995.

[4] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of SIGMETRICS/Performance '95*, pages 267–277, 1995.

[5] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall Software Series. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1986.

[6] V. H. Barocas and R. T. Tranquillo. An Anisotropic Biphasic Theory of Tissue-Equivalent Mechanics: The Interplay Among Cell Traction, Fibrillar Network Deformation, Fibril Orientation and Cell Contact Guidance. *J. Biomech. Eng.*, 119(137), 1997.

[7] V. H. Barocas and R.T. Tranquillo. Biphasic Theory and In Vitro Assays of Cell-Fibril Mechanical Interactions in Tissue-Equivalent Collagen Gels. In V. C. Mow and et al., editors, *Cell Mechanics and Cellular Engineering*, pages 185–209, New York, 1994. Springer-Verlag.

[8] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation.* The Adam Hilger Series on Plasma Physics. Adam Hilger, New York, 1991.

[9] R. H. Bisseling and W. F. McColl. Scientific Computing on Bulk Synchronous Parallel Architectures (Short Version). In B. Pehrson and I. Simon, editors, *Proc. 13th IFIP World Computer Congress*, volume 1. Elsevier, 1994.

[10] A. Bricker, M. Litzkow, and M. Livny. Condor Technical Summary. Technical Report CS-TR-92-1069, Comp. Sc. Dept, Univ of Wisconsin, Madison, Jan 1992.

[11] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk, and A. Skjellum. User's Guide to mpich, a Portable Implementation of MPI. Technical report, 1995.

[12] Gilbert Cabillic and Isabelle Puaut. Stardust: An Environment for Parallel Programming on Networks of Heterogeneous Workstations. *J. Parallel and Distributed Computing*, 40(1), Jan 1997.

[13] Radu Calinescu. Conservative Discrete-Event Simulations on Bulk Synchro-nous Parallel Architectures. Technical Report TR-16-95, Oxford University Computing Laboratory, 1995.

[14] Clemens H. Cap and Volker Strumpen. The Parform - A High Performance Platform for Parallel Computing in a Distributed Workstation Environment. Technical report, University of Zurich, Zurich, Switzerland, June 22, 1992.

[15] Clemens H. Cap and Volker Strumpen. Efficient Parallel Computing in Distributed Workstation Environments. *Parallel Computing*, pages 1221–1234, 1993.

[16] Nicholas Carriero, Eric Freeman, Gelernter, and David Kaminsky. Adaptive Parallelism and Piranha. *Computer*, 28(1):40–49, January 1995.

[17] J. D. Cavanaugh and T. J. Salo. Internetworking with ATM WANs. In William Stallings, editor, *Advances in Local and Metropolitan Area Networks.* IEEE Computer Society Press, 1994.

[18] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant. Bulk Synchronous Parallel Computing–A Paradigm for Transportable Software. In *Proc. 28th*

Hawaii Int. Conf. System Sci. (IIHICSS95), Vol. II, pages 268–275, Los Alamitos, Calif., 1995. IEEE Comp. Soc. Press.

[19] D. R. Cheriton. The V Distributed System. *Comm. of the ACM*, 31(3):314–333, March 1988.

[20] B. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. In *1997 ACM Workshop on Java for Science and Engineering Computation*, June 1997.

[21] D. R. Cox. *Renewal Theory.* Methuen's monographs on applied probability and statistics. John Wiley & Sons Inc., New York, 1962.

[22] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A Practical Model of Parallel Computation. *Communications of the ACM*, November 1996.

[23] Narsingh Deo. *Graph Theory with Applications to Engineering and Computer Science.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1974.

[24] Peter A. Dinda and David R. O'Hallaron. Statistical Properties of Host Load in a Distributed Environment. To appear in Proc. 4th Workshop on Languages, Compilers and Runtime Environments, May 1998.

[25] Jonathan M. D. Hill et al. BSPlib: The BSP Programming Library. Technical Report PRG-TR-29-9,, Oxford University Computing Laboratory, May 1997.

[26] R. Felderman, E. Schooler, and L Kleinrock. The Benevolent Bandit Laboratory: A Testbed for Distributed Algorithms. *IEEE Journal of Selected Areas in Communcation.*, 7:303–311, Feb 1989.

[27] I. Foster and S. Tuecke. Enabling Technologies for Web-Based Ubiquitous Supercomputing. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, pages 112–119, 1996.

[28] Patrick H. Fry, J. Nesheiwat, and B. K. Szymanski. Twin Primes and Brun's Constant: A Distributed Approach. Submitted to IEEE International Symposium on High Performance Distributed Computing, 1998.

[29] Al Geist and et al. PVM 3 USER's Guide and Reference Manual. Technical report, Oak Ridge National Laboratory, Oak Ridge, Tennessee, September 1994.

[30] A. V. Gerbessiotis and C. J. Siniolakis. Selection on the Bulk Synchronous Parallel model with Applications to Priority Queues. In *Proc. 1996 International Conference on Parallel an Distributed Processing Techniques and Applications (PDPTA '96)*, Sunnyvale, California, USA, August 1996.

[31] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. In O. Nurmi and E.Ukkonen, editors, *Proc. Third Scandinavian Workshop on Algorithmic Theory*, Lecture Notes in Computer Science, pages 1–18, Berlin, 1992. Springer Verlag.

[32] Mark K. Goldberg and David L. Hollinger. Database Learning: a Method for Empirical Algorithm Design. In *Proc. Workshop on Algorithm Engineering*, September 1997.

[33] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI:Portable Parallel Programming With the Message-Passing Interface.* The MIT Press, 1994.

[34] Hsieh, Jenwei, DHC Du, J. A. MacDonald, J. P. Thomas, J. T. Pugaczewski, J. Kays, and M. Wiklund. Experimental study of Extended HIPPI Connections over ATM Networks. In *Proceedings of Infocom 96*, March 1996.

[35] Jeremy Casas and Ravi Konuru and Steve W. Otto and Robert Prouty and Jonathan Walpole. Adaptive Load Migration Systems for PVM. Technical report, Oregon Graduate Institute of Science & Technology, Portland, Oregon, March 1994.

[36] David B. Johnson. Scalable support for transparent mobile host internetworking. *Wireless Networks, special issue on "Recent Advances in Wireless Networking Technology,"*, 1(3):311–321, October 1995.

[37] L. Kleinrock and W. Korfhage. Collecting Unused Processing Capacity: An Analysis of Transient Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(5), May 1993.

[38] P. Krueger and R. Chawla. The Stealth Distributed Scheduler. In *Proc. 11th Intl. Conf. Distributed Computing Systems*, pages 336–343, Arlington, 1991.

[39] David J. Kuck. *High Performace Computing: Challenges for Future Systems.* Oxford University Press, New York, 1996.

[40] P. C. Liewer and V. K. Decyk. A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes. *J. of Computational Physics*, 85:302–322, 1989.

[41] P. C. Liewer, V. K. Decyk, J. D. Dawson, and G. C. Fox. Plasma Particle Simulations on the Mark III Hypercube. *Mathematical and Computer Modelling*, 11:53–54, 1988.

[42] Litzkow. Condor Distributed Processing System. *Dr Dobb's Journal*, pages 40–48, Feb 1995.

[43] M. Litzkow. Remote Unix. In *Proceedings of 1987 Summer Usenix Conferences*, Phoenix, Arizona, June 1987.

[44] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. 8th Intl. Conf. Distributed Computing Systems*, San Jose, California, June 13-17, 1988.

[45] Mike Litzkow and Miron Livny. Experience With The Condor Distributed Batch System. In *IEEE Workshop on Experimental Distributed Systems*, Huntsville, Alabama, October 1990.

[46] J. R. Lyle and C. Lu. Load Balancing From A Unix Shell. In *Proc. 13th Conference on Local Computing Networks*, pages 181–183, Oct 1988.

[47] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proc. of ISCA 24*, Denver, Co, June 1997.

[48] Richard Miller. A Library for Bulk-synchronous Parallel Programming. In *British Computer Society Workshop on General Purpose Parallel Computing*, Dec 1993.

[49] Richard Miller and Joy Reed. The Oxford BSP Library Users' Guide, version 1.0. Technical report, Oxford Parallel, 1993.

[50] MPI: A Message Passing Interface Standard. Technical report, Message Passing Interface Forum, May 5, 1994.

[51] S. J. Mullender and et al. Amoeba: A Distributed Operating System For The 1990s. *IEEE Computer*, 23:44–53, May 1990.

[52] M. W. Mutka and M. Livny. Profiling Workstations' Available Capacity for Remote Execution. In *Proc. 12th Symposium on Computer Performance*, Brussels, Belgium, December 7-9, 1987.

[53] M. W. Mutka and M. Livny. Scheduling Remote Processing Capacity in a Workstation-Processor Bank Computing System. In *Proceedings of the 7th International Conference of Distributed Computing Systems*, pages 2–9, Berlin, West Germany, September 21-25, 1987.

[54] M. V. Nibhanupudi, C. D. Norton, and B. K. Szymanski. Plasma Simulation on Networks of Workstations Using the Bulk-Synchronous Parallel Model. In *Proc. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'95)*, Athens, Georgia, Nov 1995.

[55] M. V. Nibhanupudi and B. K. Szymanski. Adaptive Parallelism in the Bulk-Synchronous Parallel model. In *Proceedings of the Second International Euro-Par Conference*, Lyon, France, Aug 1996.

[56] M. V. Nibhanupudi and B. K. Szymanski. Adaptive Bulk-Synchronous Parallelism in a Network of Nondedicated Workstations. To appear in Proc. International Symposium on High Performance Computing Systems and Applications, HPCS'98, 1998.

[57] M. V. Nibhanupudi and B. K. Szymanski. Runtime Support for Virtual BSP Computer. In Jose Rolim, editor, *Parallel and Distributed Processing*, Lecture Notes in Computer Science, pages 147–158. Springer, 1998.

[58] D. A. Nichols. Using Idle Workstations In A Shared Computing Environment. In *Proc. Eleventh ACM Symp. Operating System Principles*, ACM, Nov 1987.

[59] C. D. Norton, B. K. Szymanski, and V. K. Decyk. Object Oriented Parallel Computation for Plasma PIC Simulation. *Communications of the ACM, 38(10)*, October 1995.

[60] C. D. Norton, B. K. Szymanski, and V. K. Decyk. Parallel Object Oriented Implementation of a 2D Bounded Electrostatic Plasma PIC Simulation. In *Proc. Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 207–212, San Francisco, California, February 15–17, 1995.

[61] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[62] Steve Rodrigues, Tom Anderson, and David Culler. High-Performance Local-Area Communication Using Fast Sockets. In *Proc. of USENIX '97*, 1997.

[63] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS, Boston, 1996.

[64] J. F. Shoch and J. A. Hupp. The ẅormP̈rograms - Early Experience With A Distributed Computation. *Communications of the ACM*, 25:172–180, Mar 1982.

[65] David B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[66] Marc Snir, Steve Otto, Steven Huss-Lederman, and David Walker. *MPI: The Complete Reference*. Scientific and Engg Computation Series. MIT Press, 1996.

[67] J. S. Steinman. Incremental State Saving in SPEEDES using C++. In *Proc. Winter Simulation Conference*, pages 687–696, Los Angeles, California, December 1993.

[68] W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall Software Series. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1990.

[69] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

[70] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

APPENDIX A
Adaptive BSP Library User Guide

A.1 The ABSP Library

The Adaptive BSP library is an implementation of the adaptive replication scheme, built on top of the Oxford BSP library. In addition to the functionality of the Oxford BSP library, the ABSP library provides functionality to achieve adaptive parallelism in environments involving transient processors. The following gives a brief summary of all the functions available in the ABSP library.

A.1.1 Process Management

The following two functions of the Oxford BSP Library are available in the ABSP library for process management.

```
bspstart(int argc, char* argv,
           int maxprocs, int* nprocs, int* mypid)
```

This function creates parallel processes on remote hosts specified by the “host file”. **maxprocs** is the maximum number of remote processes desired. The actual number of processes created is returned in the **nprocs** parameter and the process number of each process is returned in the variable **mypid**.

```
bspfinish()
```

This function terminates all remote processes and the execution continues with the original process that created the remote processes. This call must be initiated by all the participating processes.

A.1.2 Communication

The following two functions of the Oxford BSP library are available in the ABSP library for communication among the component processes.

```
bspstore(int topid, char* src, char* dst, int nbytes)
```

This function transfers **nbytes** of data from the location **src** to the location specified by **dst** in the remote process specified by the process id **topid**.

```
bspfetch(int frompid, char* src, char* dst, int nbytes)
```

This function transfers **nbytes** of data from the location **src** in the remote process specified by the process id **frompid** to the location specified by **dst** in the local process. Note that the destination in bspstore and the source in the bspfetch refer to data locations in the remote process. Thus these locations must be static or global addresses.

A.1.3 Supersteps

The following constructs of the Oxford BSP library are available in ABSP library for defining supersteps. These constructs delimit the code segment that constitutes the superstep.

```
bspsstep(int stepid)
bspsstep_end(int stepid)
```

In addition to the above, the ABSP library provides constructs to define computation supersteps that perform data replication, detection of process failures and replication of failed processes.

```
bsp_comp_sstep(int stepid)
bsp_comp_sstep_end(int stepid)
```

A.1.4 Construct to Specify the Computation State

The ABSP library provides a predefined construct to specify the computation state of a component process.

```
typedef struct BspSystemState;
```

A.1.5 Specifying Replication Data

The ABSP library provides the following construct to speicfy replication data.

```
int  bsp_replication_data(void* dataptr, long nbytes,
                          void* backup,
                          char* tag, int subscript);
```

dataptr refers to an address location representing the replication data and **nbytes** is the length of the replication data in bytes. If storage is available for replicated data (the backup), then **backup** refers to an address location for the replicated data. If **backup** is zero, then the ABSP library allocates dynamic memory for the replicated data.

A string value (**tag**) can be associated with each replication data. The replication data can subsequently be referred to by using this tag. To distinguish between components of a 2 dimensional array, an integer **subscript** can be used. For simple arrays, this value can be specified as -1.

A.1.6 Initializing and Setting the Computation State Variables

The ABSP library provides functions to initialize, to reset and to set variables holding the computation state.

```
int  bsp_init_system_state(BspSystemState* bspstate);
```

This function initializes a computation state variable. It allocates an initial amount of memory for the variable and initializes all the fields appropriately.

```
int  bsp_set_system_state(BspSystemState* bspstate,
                          char* tag, int subscript);
```

This function is used to specify replication data represented by the **tag** and **subscript** as part of the computation state represented by the variable **bspstate**.

```
void bsp_reset_system_state(BspSystemState* bspstate);
```

This function resets the computation state represented by the variable **bspstate** to nil.

A.1.7 Specifying the Computation State

The following functions specify the specific and local system state, respectively, at any point in the program execution. In a computation superstep (described in section A.1.3), the specific and local system states that are currently in effect are used for replication and local checkpointing.

```
void bsp_specific_system_state(BspSystemState* state);
void bsp_local_system_state(BspSystemState* state);
```

A.1.8 Setting Up the Replication Environment

```
void bsp_set_application_root_directory(char* rootdir);
```

This function specifies the directory to be used for creating the process id files, lock files and checkpoint files used in process replication and migration.

```
int  bsp_setup_replication_environment();
```

This function allocates dynamic storage for replication data for which no backup storage is specified.

```
void bsp_debug_print_system_state(BspSystemState* bspstate);
```

This function prints information on the replication data constituting the computation state represented by the state variable **bspstate**. This function is useful for debugging.

A.1.9 Miscellaneous

The following functions provide information on the number of processes, the process id and the host name.

```
int   BSP_id();
char* BSP_host();
int   BSP_nprocs();
```

A.2 Setting Up the Environment

Set the following environment variables in your shell startup (.bashrc or .cshrc or other) file. The BSP root directory can be obtained from the system administrator.

```
# Specify the BSP root directory;
export BSP_ROOT= < BSP root directory >
# Specify the file that contains the machine list
export BSP_HOSTFILE=~/host.list
# Set the PATH to include BSP bin directory
export PATH=$PATH:$BSP_ROOT/bin/:
# Set the MANPATH to include BSP man pages
export MANPATH=$MANPATH:$BSP_ROOT/man/:
```

The host file consists of entries denoting the hosts in the network on which remote processes can be created.

A.3 Compiling and Linking ABSP Programs

In order to use the ABSP library, the following files from the ABSP library need to be included.

```
bsp.h
bsp_client_state.h
bsp_application_decl.h
bsp_transient.h
bsp_replication.h
```

To enable the compiler access these files, include the following on the compilation command:

```
-I<BSP root directory>/src
```

To link the ABSP library, use the following in the link command on the SUN4 architecture machines:

```
        -L<BSP root directory>/lib/SUN4 -lsocket -lnsl -lbsp
```

In addition, Condor checkpoint library supplied with the ABSP distribution must be linked with the final executable. Using the Condor checkpointing library requires a special linking step. The following Makefile segment illustrates the link command using the GNU C or C++ compiler on a Sun Solaris machine.

```
CC = gcc
OBJ = <object files to be linked>
CONDOR_DIR = <Condor root directory>
SOLARIS_LIBS = /usr/local/gnu/lib/gcc-lib/sparc-sun-solaris2.5/2.7.2.2

$(CC) -nostartfiles -o <executable>          \
   $(SOLARIS_LIBS)/crti.o                    \
   $(CONDOR_DIR)/crtbegin.o                  \
   $(CONDOR_DIR)/condor_ckpt/condor_rt0.o    \
   ${OBJ}                                    \
   $(CONDOR_DIR)/condor_ckpt/libckpt.a       \
   $(CONDOR_DIR)/condor_ckpt/c_plus_alloc.o  \
   $(SOLARIS_LIBS)/crtend.o                  \
   $(SOLARIS_LIBS)/crtn.o                    \
   -lm -L<BSP root directory>/lib/SUN4 -lsocket -lnsl -lbsp
```

A sample makefile that illustrates this link step is supplied in the ABSP distribution.

APPENDIX B
Application: Plasma Simulation

Plasma simulation follows the trajectories of millions of particles in their self-induced fields. The simulation models the interactions between particles indirectly through the electric field induced by the particles at the points of a fixed grid. In the replicated grid version, particles are partitioned among the processors but the grid is replicated. All interactions between the particles and grid points take place locally on each processor. After the charge deposition operation, the grid charge is globally combined to produce a grid that is consistent across all processors. Section B.1 shows the simulation main loop in pseudocode.

B.1 Simulation Main Loop

```
for (step=0; step < Max_Simulation_Steps; step++){
  Computation Superstep:
    Add Ion Density to Charge/Density Field
    Calculate electric field on grid points;
    Initialize Charge Density;
    Initialize Kinetic Energy;
    Adavance Particle Positions and Velocity;
  Communication Superstep:
    Combine Kinteic Energy (KE)  over all processors;
  Output Energy Diagnostics; /* PE, KE and Total Energy */
  Computation Superstep:
    Compute charge deposition on grid points due to particles;
  Communication Superstep:
    Combine charge depositions on local grids over all processors;
}
```

APPENDIX C
Application: Finding the Maximum Independent Set of a Graph

The maximum independent set program uses a recursive backtracking search algorithm to search for the largest independent set. The basic recursive search algorithm is shown in pseudocode in section C.1. In the parallel version of the program, the recursive search through the virtual subgraph proceeds in an identical fashion on all processors upto a certain level. When the subgraph to be searched is of size that is below a certian threshold, processes search specific subgraphs of this subgraph and exchange information on the maximum independent set found so far at each processor.

C.1 Search algorithm

```
/*
 * Recursively search a graph;
 * The graph is represented by
 *     a list of vertices and
 *     a list of their degrees
 * The adjacency matrix is globally available;
 */
Search algorithm
{
  Create local copies of vertex and degree lists;
  Determine the number of search trees to be searched at this level;
  If the number of vertices is ZERO
    /* Graph is empty; no need to search further */
    If the size of the independent set found in the current search
      is bigger than the maximum independent set found so far,
      record this independent set as the maximum independent set.
  Else
    /* check each child subtree */
    for (position=0;position<lim;position++){
      Select the next vertex as the vertex with the lowest degree;
      Remove this vertex from the list of vertices and the list of
        degrees;
      Record vertex selected and the backtracking
        coordinate (position of the subgraph being searched);
      Delete all vertices that are adjacent to the selected vertex
        from the list of vertices and the list of degrees to create
        the vertex and degree lists for the subgraph to be searched.
      Update the degree count of the subgraph;
      Find the best independent set of the subgraph
      (Recursive call to the Search algorithm);
    }
}
```