Computer Science Masters Project

Harvesting Idle Processor Cycles on Nondedicated Network of Workstations

by

Tevfik Kosar

Rensselaer Polytechnic Institute
Troy, NewYork

July 1999
(For Graduation August 1999)

1. INTRODUCTION

The most powerful computers of today are parallel machines with thousands of processors. According to recent rankings, a machine called Janus, with 9,216 Pentium Pro processors, outperforms all of its competitors. It is certainly possible to build a machine which is more powerful than Janus by just increasing the number of processors, but when its high cost is considered, one will see that this is not an efficient solution.

On the other side, there are millions of processors in the world, connected with a global network to each other, which are idle most of the time. Recent advances in computer networking, especially in network speed, allows the use of workstations which are connected through a network as a parallel computer. If we could use a few percent of the wasted processor cycles on the Net in our computations, we would gain more computational power than any supercomputer in the world just for zero cost.

There have been several attempts to utilize the power of workstations which are connected via networks [2]. One of the most popular systems is the NOW (Network of Workstations) project at University of California-Berkeley [1]. This project aims to harness the power of clustered machines connected via high-speed networks. On April 30th, 1997, the NOW team achieved over 10 GFLOPS on the LINPACK benchmark, pushing the NOW into the top 200 fastest supercomputers in the world. This technological evolution allows NOW to support a variety of different workloads, including parallel, sequential, and interactive jobs, as well as scalable web services, including the world's fastest web search engine, and commercial workloads, such as NOW-Sort, the world's fastest disk-to-disk sort.

The network of workstations used in this project is a dedicated network. There are also other attempts which try to utilize non-dedicated idle workstations for computations. One of the most successful such systems is the Condor project [8], developed by Miron Livny and his colleagues at the University of Wisconsin-Madison. Condor automatically locates workstations which are idle and transfers jobs to them. The jobs are periodically checkpointed and migrate from machine to machine until they are completed. Studies of Condor showed that jobs submitted to it made use of over 180 CPU-days per week of otherwise wasted machine cycles. Condor is now running at more than a dozen universities and other sites.

Another approach is proposed by Mohan V. Nibhanupudi and Boleslaw K. Szymanski [16] at Rensselaer Polytechnic Institute. This approach treats the temporary unavailability of a workstation as a transient failure and tries to reduce its affect on the execution time of the program. It relies on the recovery of the computations of a failed process by replicating its computations on another available workstation and 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. This approach is referred as the Adaptive Replication Scheme (ARS) [14].

In our project, we analyzed the performance of parallel computations on dedicated and non-dedicated network of workstations. In our analysis, we used the application program GALE, a simulation of evolution in epidemics, which was previously implemented in MPI by W. Maniatty [13]. We first implemented GALE using the Oxford BSP Library on dedicated network of workstations (see Chapter 4). Then, we ported GALE to non-dedicated network of workstations using the Adaptive Replication Scheme (see Chapter 5), and made performance comparisons with the dedicated NOW implementation. Finally, we also implemented GALE using the latest version of BSPlib (v1.4), and compared the test results with the old BSP version (see Chapter 6). We give general information about the BSP computational model and the application program GALE in chapter 2 and chapter 3 respectively. In chapter 7, we discuss the results and propose possible improvements that can be done on the existing models.

2. BULK SYNCHRONOUS PARALLELISM

The Bulk Synchronous Parallel (BSP) Model [12] is initially proposed by L. G. Valiant and further developed by W. F. McColl and R. Miller. BSP Model brings a wide range of hardware and software architectures together into a single abstraction of a parallel computer by playing the same part as the traditional Van Neuman model for sequential computation.

Bulk Synchronous Parallelism is a style of parallel programming developed for general purpose parallelism, that is parallelism across all application areas and a wide range of architectures. Portability of the primitive parallel operations is achieved by basing their semantics on the architecture-independent model of parallel computation

2.1. Why BSP?

We selected the Bulk Synchronous Parallel (BSP) Model for implementation in our project among a huge number of parallel computation models, since most of these models are inadequate either in portability or in performance.

The parallel computation models which are based on message passing are inadequate because of the complexity of correctly creating paired communication actions (send and receive) in large and complex programs. Such systems can easily result in deadlocks and the prediction of their performance is very hard because of the interchange of large numbers of individual data transfers [19].

The models which are based on shared memory are easier to program since they provide the concept of a single, shared address space and so a whole class of placement decisions are avoided. But this requires either careful crafting of programs, in the PRAM style, or expensive lock management. Implementing shared memory programs requires a larger fraction of the computers' resources to be dedicated to communication and the maintenance of coherence. In additions, these systems are more expensive than others are.

On the other hand, the BSP model is simple to write, efficient and fully portable. It is applicable across the whole spectrum of general-purpose architectures and provides efficient, scalable and predictable performance on a given architecture.

2.2. The BSP Computational Model

The BSP model is proposed as a bridge between software and the underlying architecture. It plays exactly the same part as the traditional Van Neumann model for sequential programming: a unifying abstraction of a general purpose computer, in which a variety of high-level languages can be used for implementation and the applications are independent of the target architecture.

R. Miller and J. Reed [13] present a slightly simplified version, in which the components of a BSP computer consist of:

- a set of sequential processors each with local memory
- a communication system which the processors can use to access non-local data
- a mechanism to globally synchronize the processors

The term *bulk-synchronous* corresponds to a model between the extreme points of synchronous and asynchronous parallelism. In a fully synchronous system, a global clock synchronizes the processors on every instruction. On the other hand, in a fully asynchronous system, processes can work independently at different speeds, and are synchronized inherently in pairs only when a communication takes place. Bulk-synchronous parallelism merges the two models in some sense. The processors of a bulk-synchronous parallel computer progress together through a program, but the period of synchronization is a superstep: a fragment of a computation during which each processor performs a sequence of operations using only its own local data, and may also initiate requests to read and wait in a global barrier synchronization until the non-local data exchanges are completed, before proceeding to the next superstep.

A particular BSP machine has p processors each of which can execute s floating-point operations per second, and a communications system which has a global capacity of b words per second. L is the minimum amount of time which must elapse between successive global synchronizations. The cost of global communication is expressed by the BSP machine parameter g. The accuracy of the model requires L to vary with p, and g to vary with data transfer block size [18].

The BSP approach is to use cost of an *h-relation* which is a global communication pattern in which every processor can send and receive up to a total of h units of data and constitutes a uniform cost model for non-local memory access.

With these assumptions, a superstep has a time cost C which satisfies:

$$C \leq X + L + gh \tag{1}$$

in which X is the maximum number of local operations performed by any processor.

3. THE APPLICATION: GALE

Population dynamics and epidemics spread are of great significance for agriculture, ecosystems and society [7]. Biologists and ecologists have broadly studied these fundamental phenomena to achieve pest and disease control.

Scientists believe that organisms have a *genotype* inherited from their parents which coordinates the development of the organism, and is apparent as the *phenotype* - the physical results of these developments [10]. The elimination of a species or genotype from an ecosystem is non-reversible and ecologically very significant. Evolution is governed by the process of *selection* which favors well adapted genotypes over inferior genotypes, and by *mutation* which introduces new genotypes.

Some computational challenges arose in the process of representing the underlying entities (e.g. the organisms and the habitat) and their attributes in the model (e.g. space, time, fitness and genotype data) and the resulting simulation. Historically, spatially explicit models of populations have been computationally intensive and have only recently become tractable for modeling using high performance computing.

Biological research objectives are investigation of the following problems:

- How does the range of interactions between evolutionary stable organisms influence competition/selection at the host level?
- How does the range of interaction between evolutionarily stable host organism influence the spread/persistence caused by a unique pathogen genotype?
- How does the range of interaction between evolutionarily stable hosts influence the interaction between two competing genotypes of pathogens?
- How does the range of interaction between evolving species influence competition between hosts?
- How does the range of interaction between coevolving hosts and pathogens influence the outcome of epidemics?

3.1. Some definitions

In an epidemic, hosts are assumed to reside in a *habitat*. The hosts are subject to *infection* by a *pathogen* which causes disease. For certain diseases *parasites* may infect hosts, and cause/spread the disease. A host which does not have the disease, but may catch the disease is said to be *susceptible*. A host which has the disease, and may spread it to other hosts is said o be *infective*.

3.2. The GALE Model

A four-species community is assumed and the host species are identified by a binary variable $h \in \{0, 1\}$. h=1 implies "good" host (from the perspective of the selective parasite), and h=0 implies "bad" host. The parasite can expect to live longer and more readily acquire new hosts when it exploits a good host. The parasite acts as a vector, transmitting a pathogen from infected hosts to uninfected hosts. The ecosystem is assumed to be closed to immigration, so that extinction of one or more species is possible.

The model partitions the environment into a regular lattice of J sites. Each site k is in one of nine possible states. A particular site may be empty (state 1), or may be occupied by one of the host species, but not both. Table 3.1 shows the possible states and their configurations and Figure 3.1 shows the transitions between states [9]. In the parasite and pathogen columns, 0 represent *absent* and 1 represents *present*.

state	host	Parasite	Pathogen
1	absent	0	0
3-h	h	0	0
5-h	h	1	0
7-h	h	0	1
9-h	h	1	1

Table 3.1: Ecological States

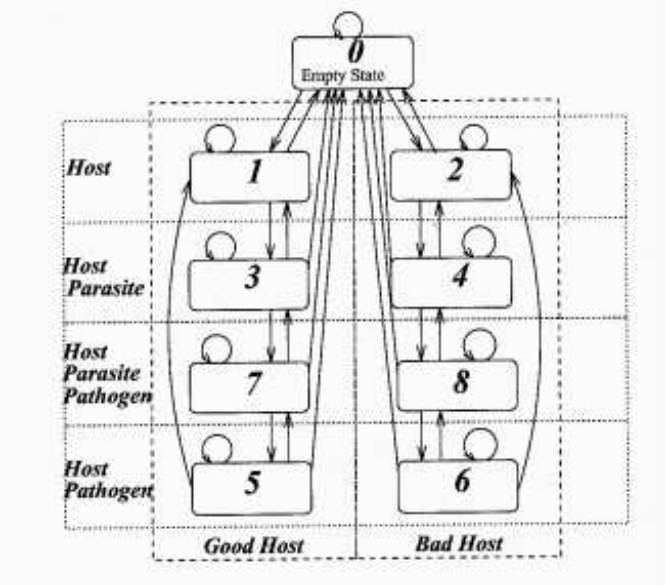

Figure 3.1: Ecological State Transition Diagram

The underlying model of GALE [11] has localized interactions, so a static block data decomposition was selected. To compute state transition probabilities, GALE must traverse the set of sites within the interaction neighborhood, rather than to count the number of sites in each state.

To model reproduction, all genotypes that can reach a given site must be known, so GALE must maintain a dynamic set of eligible genotypes. Set union, intersection and difference operations have a cost proportional to the cardinality of the sets being merged. As a result, the state transition function computed at each simulation step is very complex, since it involves operations on the multi-set of genotypes in the interaction neighborhood. This results in great dependence of performance on the area of the interaction neighborhood. Possible improvements include use of segmentation of the environment according to sites genotypes or classes of genotypes.

The processor which owns the partition in which a particular site resides is responsible for computing its next state, and therefore requires information about the current state of boundary sites on neighboring processors (since some stencils can span partitions). State information which can be efficiently computed locally and does not need to be computed across partition boundaries is maintained for each site.

The Gale model was originally implemented using C++ and MPI by W. Maniatty on a shared memory architecture [11].

4. NOWS USING BSP

We used the Oxford BSP Library [13], developed by Richard Miller and Joy Reed, to implement GALE on dedicated Network of Workstations. The Oxford BSP Library is based on a simplified version of the BSP Model and consists of six primitive routines. For convenience and efficiency, the library also provides higher level routines which are semantically defined in terms of these six primitive routines. Table 4.1 shows the Oxford BSP Library routines.

Process Management	Bsp_start(int max_procs, int* num_procs, int* my_pid)
	Bsp_finish()
Synchronisation	Bsp_sstep(int stepno)
	Bsp_sstep_end(int stepno)
Communication	Bsp_fetch(int from_pid, void* from_data, void* to_data, int nbytes)
	Bsp_store(int to_pid, void* from_data, void* to_data, int nbytes)
High-level Templates	Bsp_broadcast(int from_pid, void* from_data, void* to_data, int nbytes)
	bsp_reduce(reduce_routine(), void* from_data, void* to_data, int nbytes)

Table 4.1: The Oxford BSP Library Routines

These routines are used in the implementation of the Adaptive BSP (A-BSP) Library by M. Nibhanupudi. For this reason, we also used the same routines to implement GALE on dedicated network of workstations, so that we could make a better analysis and adequate comparisons between two systems.

4.1. Implementation

We implemented GALE on a network of 20 Sun workstations in the Computer Science Department of Rensselaer Polytechnic Institute. Solaris 5.6 is installed on each of the workstations and they are connected to each other with a 10 Mbps Ethernet. Most of the workstations used are in the graduate laboratory of the department and open to the public use 24 hours a day, whereas a few of the used machines are in the graduate student offices and generally used only by their owners. Table 4.2 shows the configuration of the machines that constitute our NOW.

Quantity	Machine type	Memory
1	Sun UltraSparc II	384M
1	Sun UltraSparc II	192M
1	Sun UltraSparc I	192M
5	Sun SparcStation 20	128M
1	Sun SparcStation 10	128M
1	Sun SparcStation 10	40M
2	Sun SparcStation 5	64M
5	Sun SparcStation 5	40M
1	Sun SparcStation 5	32M
2	Sun SparcStation 4	64M

Table 4.2: Configuration of our NOW

The network of workstations that we used was quite heterogeneous in terms of both the capacities of the machines and their load averages throughout the day. A list of the workstations that are used in our cluster, their flop rates and loads inquired by the *bsploa*d utility of BSPlib can be found in Figure 6.2 of chapter 6.

A human readable parameter file which is parsed at runtime is used in the tests. One of the sample parameter files that we used in the tests is shown in Figure 4.1. In he MPI version of GALE, the parameter file was read by all of the processors concurrently using the features of the distributed file system NFS on which our NOW is based. In the BSP version, we did not prefer the same method. In our system, only the master processor reads the parameter file and then all of the other processors fetch the data from the master processor in a single step. All of these operations are performed in one superstep. In all of the I/O operations we applied these method.

The tests are run for different input sizes. For small input sizes the communication over computation ratio was larger, which caused a high latency and the running time did not scale well with the increase in the number of processors. To get better results we should increase computation over communication ratio, so we tried for larger input sizes.

```
0     #         - Version Number of parametter file layout
1.0   # mu[1]   - Base (Unadjusted or raw) Host death rate 1
2.0   # mux[1]  - x value used to adjust death rate 1
3.0   # muy[1]  - y value used to adjust death rate 1
4.0   # mu[2]   - Base (Unadjusted or raw) Host death rate 2
5.0   # mux[2]  - x value used to adjust death rate 2
6.0   # muy[2]  - y value used to adjust death rate 2
7.0   # mu[3]   - Base (Unadjusted or raw) Host death rate 3
8.0   # mux[3]  - x value used to adjust death rate 3
9.0   # muy[3]  - y value used to adjust death rate 3
10.0  # mu[4]   - Base (Unadjusted or raw) Host death rate 4
11.0  # mux[4]  - x value used to adjust death rate 4
12.0  # muy[4]  - y value used to adjust death rate 4
13.0  # mu[5]   - Base (Unadjusted or raw) Host death rate 5
14.0  # mux[5]  - x value used to adjust death rate 5
15.0  # muy[5]  - y value used to adjust death rate 5
16.0  # mu[6]   - Base (Unadjusted or raw) Host death rate 6
17.0  # mux[6]  - x value used to adjust death rate 6
18.0  # muy[6]  - y value used to adjust death rate 6
19.0  # mu[7]   - Base (Unadjusted or raw) Host death rate 7
10.0  # mux[7]  - x value used to adjust death rate 7
21.0  # muy[7]  - y value used to adjust death rate 7
22.0  # mu[8]   - Base (Unadjusted or raw) Host death rate 8
23.0  # mux[8]  - x value used to adjust death rate 8
25.0  # muy[8]  - y value used to adjust death rate 8
26.0  # muPb    - base probability of recovery from parasite
27.0  # muPx    - x coefficient in host recovery from parasite
28.0  # muPy    - y coefficient in host recovery from parasite
29.0  # mutation_rate[0] - Bernoulli prob. per bit of mutation 0
30.0  # mutation_rate[1] - Bernoulli prob. per bit of mutation 1
31.0  # mutation_rate[2] - Bernoulli prob. per bit of mutation 2
32.0  # mutation_rate[3] - Bernoulli prob. per bit of mutation 3
33.0  # pc         - prob. of winning site competition betwen hosts
34.0  # pfb[0]     - base probability of fathering offspring 0
35.0  # pfb[1]     - base probability of fathering offspring 1
36.0  # rho[0]     - Base (Unadjusted or raw) host fecundity 0
37.0  # rhox1[0]   - adjustment for host fecundity 0
38.0  # rho[1]     - base (Unadjusted or raw) host fecundity 1
39.0  # rhox1[1]   - adjustment for host fecundity 1
40.0  # gamma_p    - base exposure chance of parasite
41.0  # gamma_px   - adjustment for parasite exposure chance
42    # max_time   - number of time steps - length of simulation
43    # steps_per_sample            - frequency of sampling
44    # host_reproductive_frequency - expressed in ``ticks''
45    # left       - left of stencil
46    # right      - right of stencil
46    # bottom     - bottom of stencil
47    # top        - top of stencil
48    # wrap flag - 1 = toroidal wrap is on, 0=off
```

Figure 4.1: Sample Parameter File

When we increased the input size, the performance improvement with regard to the number of processors started to be seen more apparently. So, we have chosen the largest possible input size that was possible in the tests and it was 960x960. We could not increase the input size more, since it required a huge memory which was above the limits of the workstations in our NOW. Even with this input size we had some difficulties especially during the peak hours. GALE required 17 MB memory per processor with this input size, and when we consider that there were workstations with only 32 MB of memory most of which is used just by the system processes, we could not run our program with full capacity even during some non-peak hours since memory of several machines was not enough.

4.2. Test Results

The execution times of GALE with regard to the number of processors on dedicated network of workstations are shown in Table 4.3 and plotted in figure 4.2. Figure 4.3 shows the plot of speedup vs. the number of processors. The speedup is defined as a ratio of a sequential implementation's run time to a parallel implementation's run time.

# of Processors	2	3	4	6	8	16	20
Execution Time (sec)	453	357	267	229	179	150	132

Table 4.3: Execution Time vs. Number of Processors on dedicated network of workstations using the Oxford BSP Library

As it can be seen from the test results, execution time and speedup scale well with regard to the number of processors, but especially for large number of processors it is far below the optimum values. Obviously this is mostly due to the high latency in communication between processors since the speed of the network that connects the workstations of our NOW is very slow. We think that if we had used a faster network like 100 Mbps Ethernet or ATM, the measured results would be very near to the optimum values.

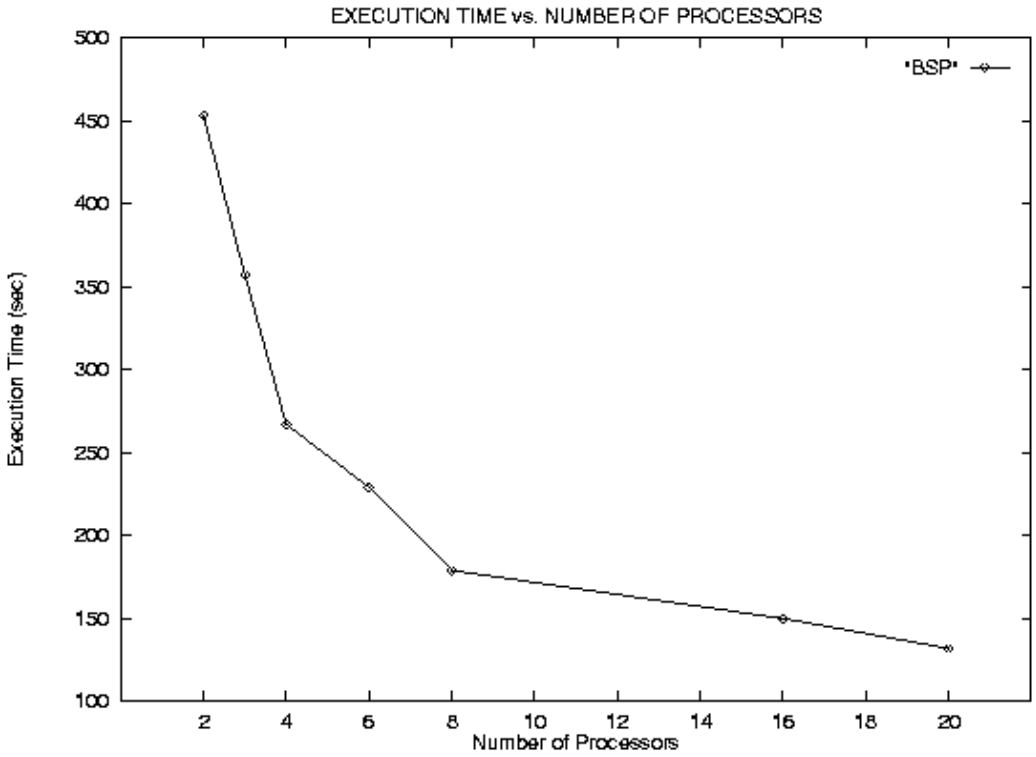

Figure 4.2: Execution Time vs. Number of Processors

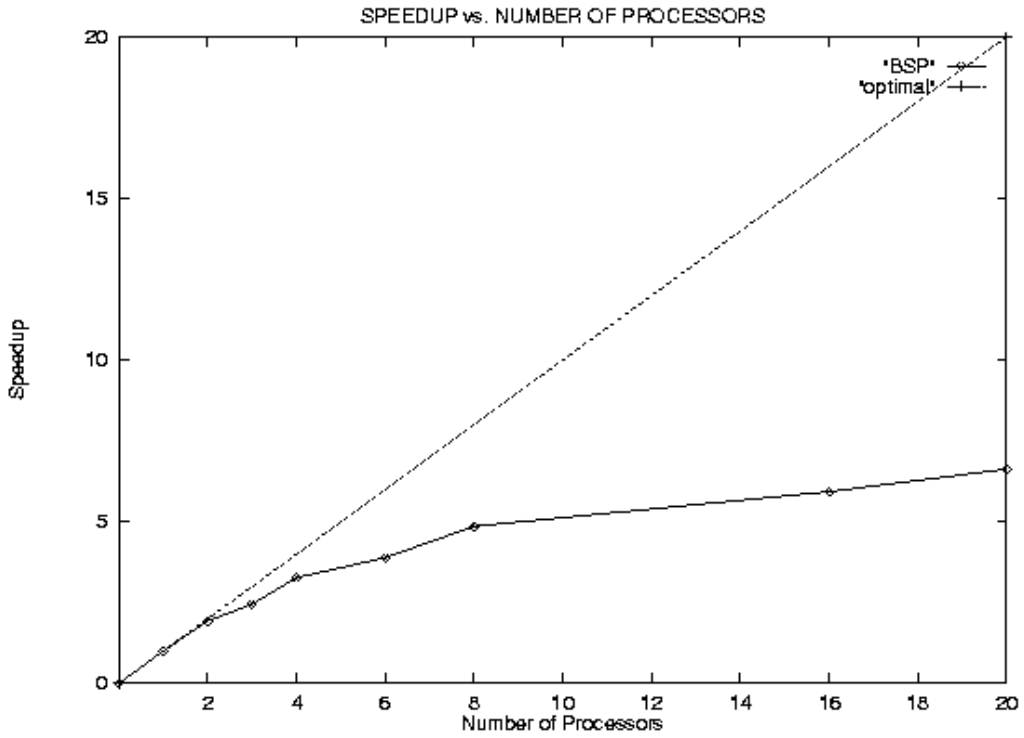

Figure 4.3: Speedup vs. Number of Processors

The source code for the main module and the communication routines (Gale.C and BSP_datacomm.C) of the dedicated network of workstations implementation of GALE using the Oxford BSP Library is given in Appendix A. The complete source code can be found at the web-site http://www.cs.rpi.edu/kosart/gale-BSP/BSP/.

5. NON-DECICATED NOWS USING A-BSP

In the implementation of GALE on non-dedicated network of workstations, we used the Adaptive Replication Scheme (ARS) [16] and the Adaptive BSP library (A-BSP) developed by M. Nibhanupudi and B. K. Szymanski.

ARS treats the temporary unavailability of a workstation as a transient failure and seeks to reduce its affect on the execution time of the application. It relies on recovering the computations of a failed process by replicating its computations on another available workstation and migration of the recovered process to a new available workstation.

A workstation is considered unavailable if there is a user logged on the system or the average CPU usage is above a threshold value, and available (idle) all other times. The unavailability of a processor is considered as a *failure* and a transition of a host processor from available to unavailable state is referred as a *transient failure* of the component process.

In the adaptive replication scheme, the computation state is eagerly replicated on a neighbor processor at the beginning of a computation step. In the event of a failure of the sender processor, the receiver processor uses the state data it received to replicate the computations of the failed processor.

ARS assumes that one of the processes is on a host owned by the user and therefore it is exempt to transient failures [15]. This process is referred as the *master process*. The master process coordinates recovery from transient failures without replicating for any of the failed processes. 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. In a computation superstep, each process in the ring sends its computation state to one or more of its successors, called *backup processes*, before starting its computation. Each process also receives the computation state from one or more of its predecessors.

When a process finishes its computation, it sends a message indicating successful completion o 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 time-out period is interpreted as the failure of the predecessor. This process then

creates a new process 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. In order to that, the restoring process uses the computation state received from that predecessor. The process than performs synchronization for itself. Each of the newly created processes performs the computation on favor of a failed predecessor and performs synchronization on its part to complete the computation step.

5.1 The Adaptive BSP Library

The Adaptive BSP Library, developed by M. Nibhanupudi and B. K. Symanski, is based on the Oxford BSP Library. It supports the primitive routines of the Oxford BSP Library and has also some extensions to it. 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. Figure 5.1 shows the adaptive extensions to the Oxford BSP Library.

Extensions to specify a computation superstep:
- bsp_comp_sstep(int sstepno);
- bsp_comp_sstep_end(int sstepno);

Extensions to specify replication data and allocate storage:
- bsp_replication_data(void* data, long nbytes, void* store, char* ttag, int subscript);
- bsp_setup_replication_environment();

Extensions to specify the 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);

Extension for recovery:
- void RecoveryFunction();

Figure 5.1: Adaptive Extensions to the Oxford BSP Library

5.2. Implementation & Results

We implemented the non-dedicated network of workstations version of GALE using the dedicated network of workstations version, which we implemented before with BSP, and the adaptive extensions to the Oxford BSP Library (A-BSP Library).

Replicating the computations of a failed process is made by eagerly saving the computation state of each process at the beginning of the computational superstep. The computation state consists of two parts, the Specific System State and the Common System State. In the Specific System State of GALE all of the data that is needed to solve the interior and boundary stencil partitions corresponding to that processor is saved. This state is distinct in each component process and is eagerly saved in each iteration of the main simulation loop since it changes in each iteration. The Common System State of GALE consists of the parameter record which is required to initialize the computations on each processor. This state is not changed during the computational superstep, and therefore it is saved only once at the beginning of the simulation. Dynamic storage is allocated to the replication data using the bsp_replication_data construct in both CURR_STATE and the PARM_REC BspSystemStates.

The execution times of GALE with regard to the number of processors on non-dedicated network of workstations are shown in Table 5.1 and plotted in figure 5.2. Figure 5.3 shows the he plot of speedup vs. the number of processors.

# of Processors	3	4	6	8	16	20
Execution Time (sec)	777	695	639	609	576	557

Table 5.1: Execution Time vs. Number of Processors in the non-dedicated Network of Workstations version of GALE

As it can be seen from the test results, execution time and speedup seems to scale well with regard to the number of processors but when we check the values more carefully, we see that for 20 processors we get a speedup of only 2 which is far below the expected results. The high latency in communication due to slow network speed is one of the reasons of these results. But we think that he most important factor

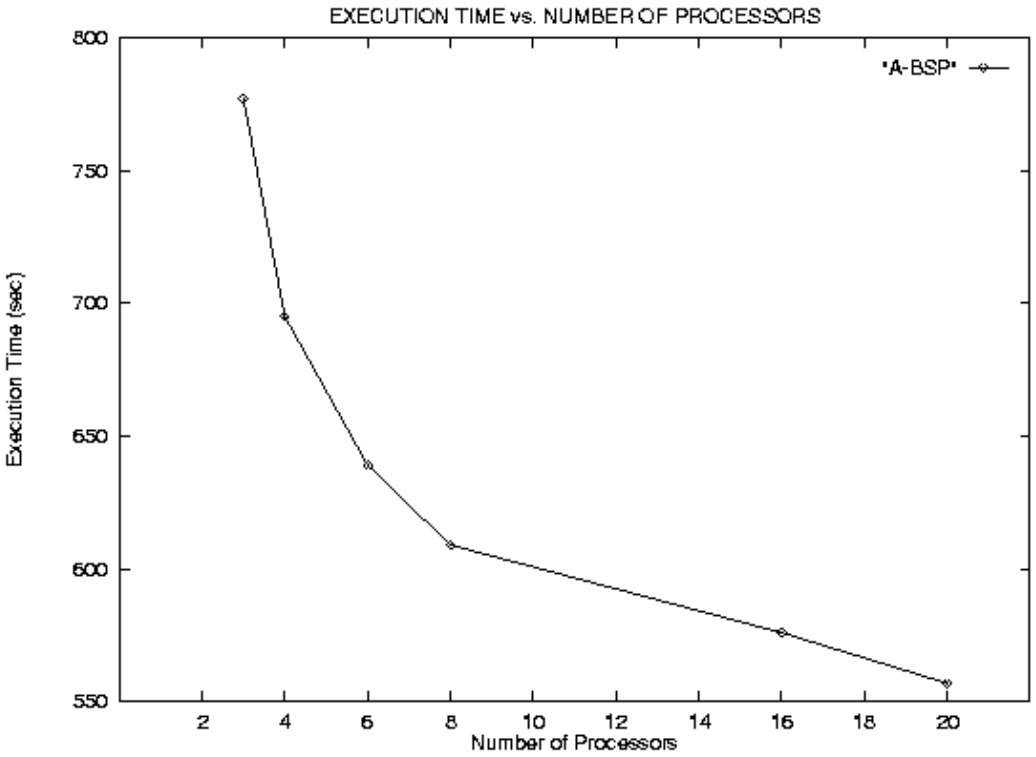

Figure 5.2: Execution Time vs. Number of Processors

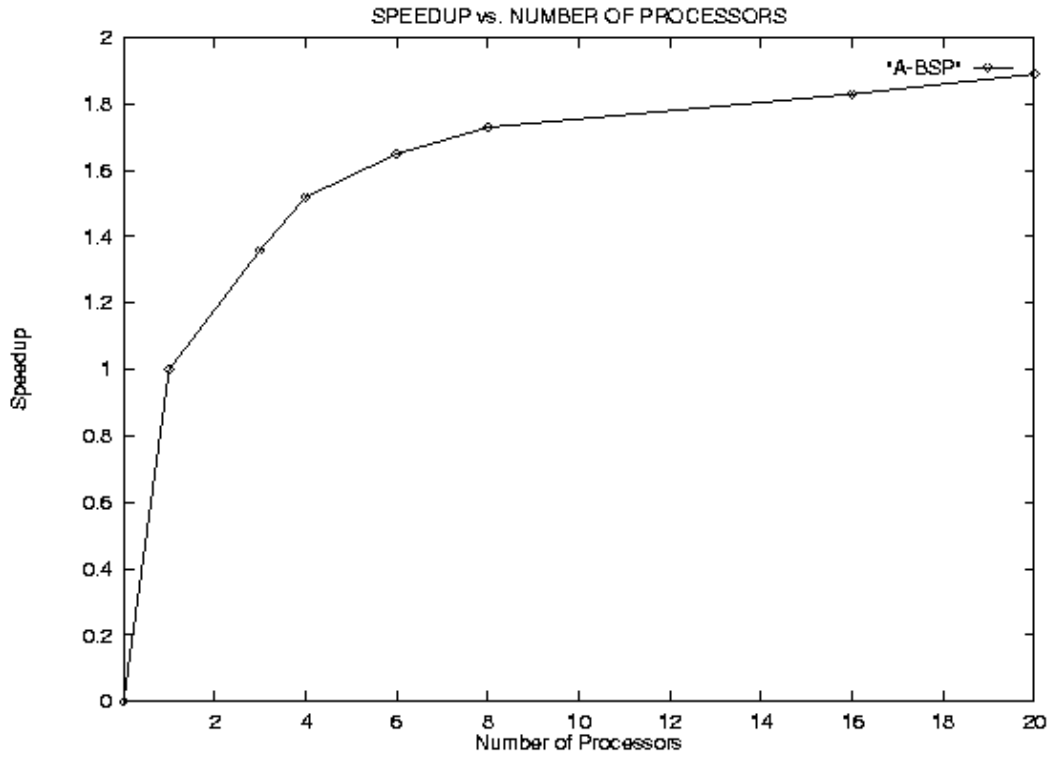

Figure 5.3: Speedup vs. Number of Processors

is that the workstations are non-dedicated and are highly utilized by other users during test hours, which in turn causes transient failures and slows down the computations.

When we compare the performances of the dedicated and non-dedicated versions of GALE (Figure 5.4), we see that whereas for 3 processors, the dedicated version is only 2.2 times faster than the non-dedicated version, for 20 processor the ratio grows up to 4.2. The reason is that we have only 20 processors in our host list, and if one of the processors fails, that means either another user starts to use that machine or the system load exceeds a threshold value – which happens most of the time, the execution time of the parallel program with high level of parallelism increases notably. But still the difference is not very high when we consider that the non-dedicated network of workstations uses only idle processor cycles and costs for zero.

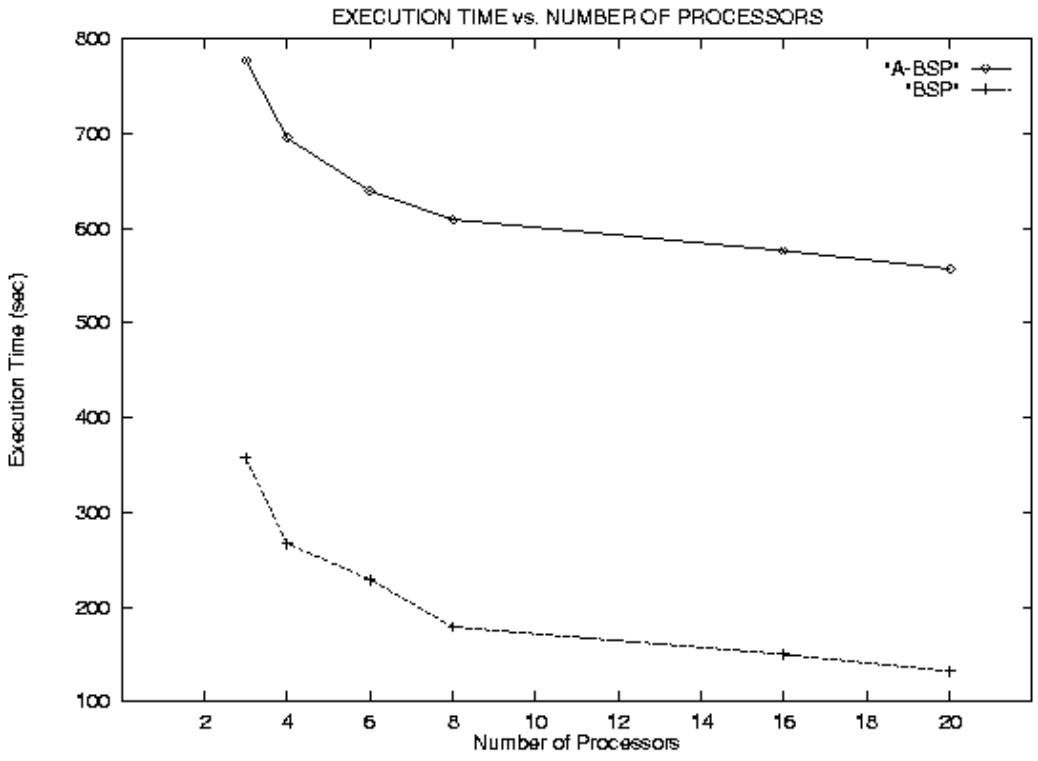

Figure 5.4: Comparison of dedicated (BSP) and non-dedicated (A-BSP) NOW versions of GALE

The source code for the main module and the communication routines (Gale.C and BSP_datacomm.C) of the non-dedicated network of workstations implementation of GALE using the Adaptive Replication Scheme and the A-BSP Library is given in Appendix B. The complete source code can be found at the web-site http://www.cs.rpi.edu/kosart/gale-BSP/A-BSP/ .

6. NOWS USING BSPLIB

We implemented GALE on dedicated network of workstations using the latest version of BSPlib [6] to compare its performance with the previous version.

The Oxford BSP toolset (v1.4) implementation of BSPlib is based on top of a variety of lower level messaging layers. The distribution provides four general classes of implementations for the following types of machines:

- Distributed memory machines where the implementation uses either proprietary message passing libraries or MPI
- Distributed memory machines where the implementation uses primitive one sided communication (e.g. the Cray SHMEM library of the T3E)
- Shared memory multiprocessors where the implementation uses either propriety concurrency primitives or System V semaphores
- Network of Workstations where the implementation uses TCP/IP or UDP/IP.

6.1. BSPlib for Network of Workstations

In building BSPlib for a network of workstations [5], either TCP/IP based message passing (MPASS_TCPIP) or UDP/IP based message passing (MPASS_UDPIP) devices can be used. The UDP/IP implementation is designed for low latency communication, and performs particularly well in a switched 100Mbps Ethernet environment [3]. However, the correct installation of the UDP/IP version depends upon setting accurate values for the message round-trip time and the inter-packet latency during the configuration process. If one is unsure of the correct values of these parameters for his network, then it is advisable to use the MPASS_TCPIP implementation of the library, as it is flexible to incorrect values of these parameters.

Before a job can be executed on a network of workstations, a number of daemons need to be started on each of the machines that form the cluster. These daemons control the access, startup, and cleanup of programs on remote nodes. There are three daemons that the user has to be aware of:

- *bspnowd*: controls the access to a machine for a particular user
- *bspportd*: maintains the correspondence between port numbers and BSPlib user daemons.
- *bsploadd*: used to ensure that BSP jobs are started on the least loaded nodes in a network.

When a BSP job requests *p* processes, the *p* least loaded machines are chosen automatically, which provides a load balancing facility at process startup time.

6.2. Implementation & Results

We selected TCP/IP based message passing (MPASS_TCPIP) device in the implementation of the dedicated NOW version of GALE since we had a high latency network (10 Mbps Ethernet). But when we run our parallel program, we observed that the workstations could not communicate with each other. It took a long time before we could finally find the reason of this problem. The TCP/IP socket low water marks were not working in our system and since BSPlib requires the SO_SNDLOWAT option to setsockopt to work correctly, the TCP/IP version was not working correctly in our network. Figure 6.1 shows the warning message that we got from BSPlib.

```
checking if send low water marks on sockets are supported by
TCP/IP... no

configure: warning:
*********************************************************************
***  The TCP/IP socket low water marks are not working on this    ***
***  platform. As BSPlib requires the SO_SNDLOWAT option to       ***
***  setsockopt to work correctly, the TCP/IP version may not     ***
***  work correctly on this platform (on some platforms this      ***
***  causes deadlock at very high loads and on others the         ***
***  TCP/IP library may not work at all --- the behaviour         ***
***  depends on the particular implementation of the protocol     ***
***  stack and/or the sockets interface). On these platforms,     ***
***  we suggest using the UDP/IP version of BSPlib.               ***
*********************************************************************
```

Figure 6.1: Warning message from BSPlib

Then we tried with the UDP/IP based message passing (MPASS_UDPIP) device instead of TCP/IP and it worked without any problem. Figure 6.2 shows the list of workstations used in our cluster, their flop rates and loads inquired by the *bsplоad* utility of BSPlib.

```
Machine           | CPUs |Nice|   Flop rate    | Load |         Age
------------------+------+----+----------------+------+-------------
einstein.cs.rpi.  |    1 |  0 | 39.9 Mflop/s | 0.09 |   6 min 32 sec
fork.cs.rpi.edu   |    1 |  0 |  9.5 Mflop/s | 0.05 |   7 min 14 sec
hash.cs.rpi.edu   |    1 |  0 |  9.4 Mflop/s | 0.05 |   7 min 33 sec
glob.cs.rpi.edu   |    1 |  0 |  9.2 Mflop/s | 0.04 |   2 min 10 sec
bullpen.cs.rpi.e  |    1 |  0 |  8.9 Mflop/s | 0.07 |   7 min 40 sec
griddle.cs.rpi.e  |    2 |  0 |  8.6 Mflop/s | 0.06 |   4 min 43 sec
trabant.cs.rpi.e  |    1 |  0 |  9.1 Mflop/s | 0.11 | 11 hr  23 min
hotplate.cs.rpi.  |    2 |  0 |  8.7 Mflop/s | 0.15 |   5 min  0 sec
module.cs.rpi.ed  |    1 |  0 |  6.3 Mflop/s | 0.05 |   2 min 53 sec
saladshooter.cs.  |    1 |  0 |  6.4 Mflop/s | 0.07 |   4 min 33 sec
achilles.cs.rpi.  |    1 |  0 |  6.2 Mflop/s | 0.04 |   2 min  0 sec
troi.cs.rpi.edu   |    1 |  0 |  6.2 Mflop/s | 0.05 |   6 min 31 sec
volga.cs.rpi.edu  |    1 |  0 |  6.3 Mflop/s | 0.08 |   7 min 21 sec
rolls.cs.rpi.edu  |    1 |  0 |  2.8 Mflop/s | 0.05 |   5 min 22 sec
juicer.cs.rpi.ed  |    2 |  0 |  8.5 Mflop/s | 1.01 |   7 min 32 sec
fridge.cs.rpi.ed  |    2 |  0 |  8.2 Mflop/s | 1.02 |   4 min 39 sec
icemaker.cs.rpi.  |    2 |  0 |  8.6 Mflop/s | 1.02 |   4 min 39 sec
eggbeater.cs.rpi  |    1 |  0 | 25.2 Mflop/s | 1.08 |         1 sec
coffeepot.cs.rpi  |    2 |  0 | 30.2 Mflop/s | 1.36 |   5 min 53 sec
dishwasher.cs.rp  |    2 |  0 | 29.8 Mflop/s | 2.19 |   4 min 12 sec
```

Figure 6.2: The list of workstations used in our cluster, their flop rates and loads inquired by *bsplоad*

The execution times of the BSPLIB implementation of GALE with regard to the number of processors on dedicated network of workstations are shown in Table 6.1 and plotted in figure 6.3. Figure 6.4 shows the plot of speedup vs. the number of processors.

# of Processors	2	3	4	6	8	16	20
Execution Time (sec)	359	276	213	171	142	123	107

Table 6.1: Execution Time vs. Number of Processors in the dedicated NOW with BSPLIB version of GALE

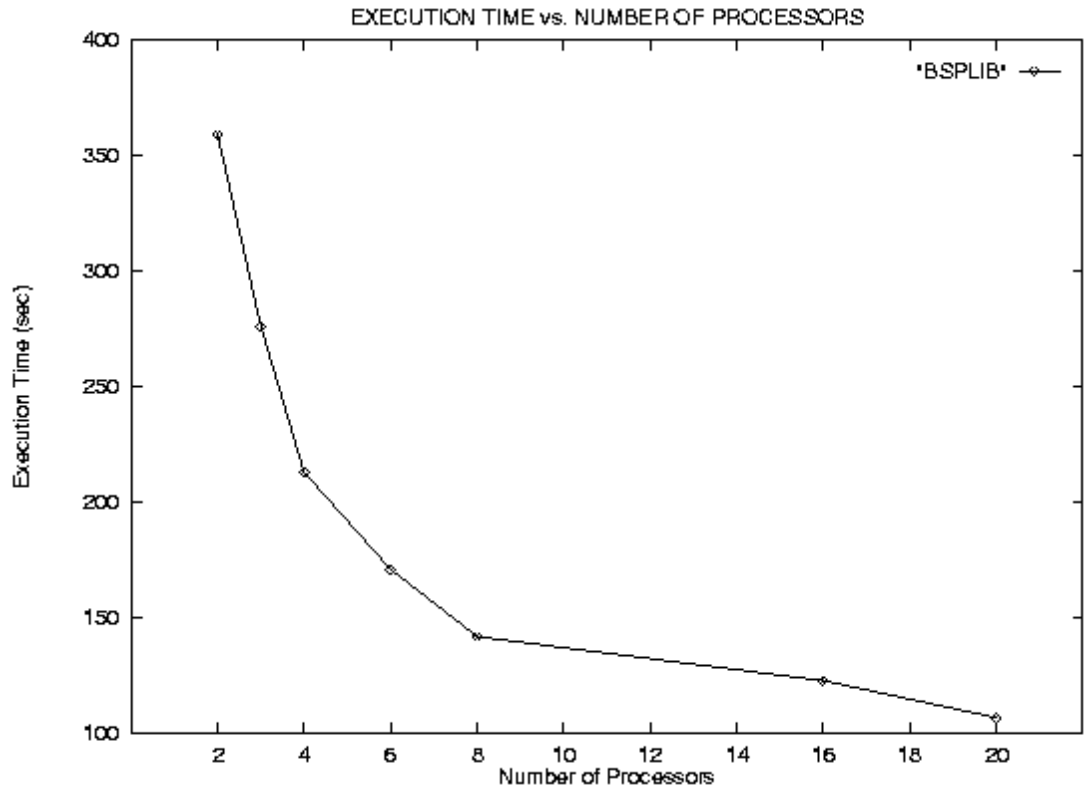

Figure 6.3: Execution Time vs. Number of Processors

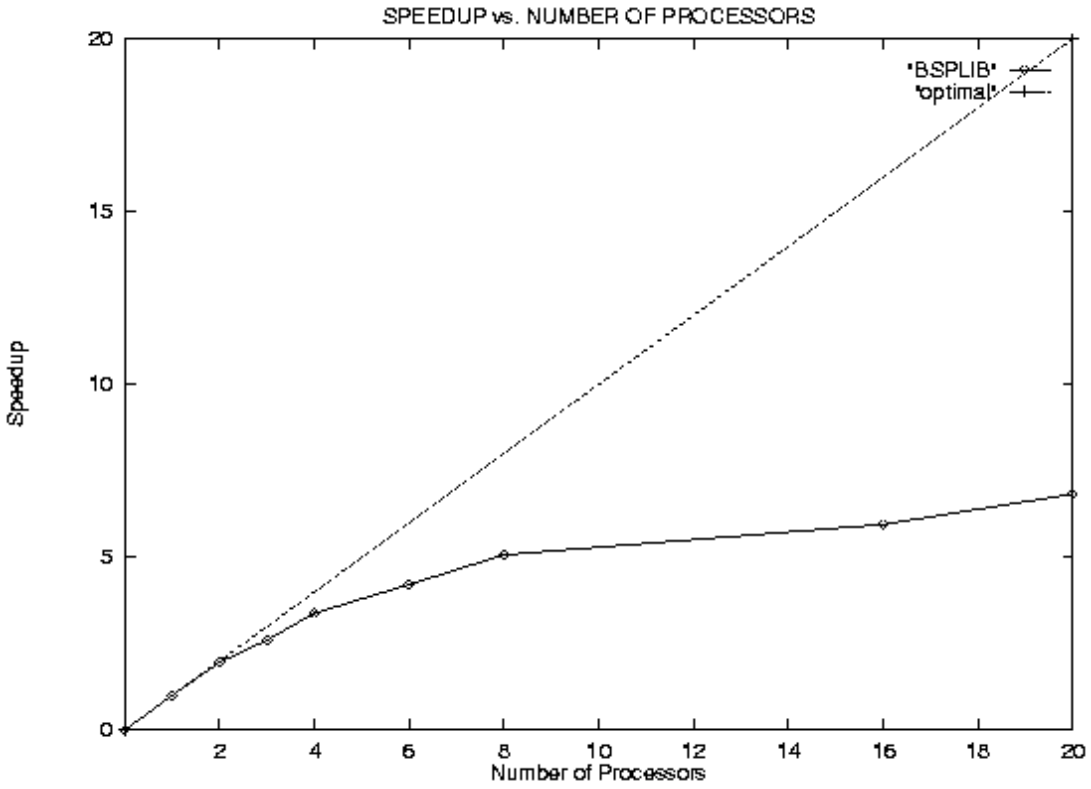

Figure 6.4: Speedup vs. Number of Processors

When we compare the performances of the latest version of BSPLIB (v1.4) and the old version of Oxford BSP Library on dedicated network of workstations (Figure 6.5), we see that the latest version of BSPLIB shows about 4/3 times better performance than the old version of the Oxford BSP Library.

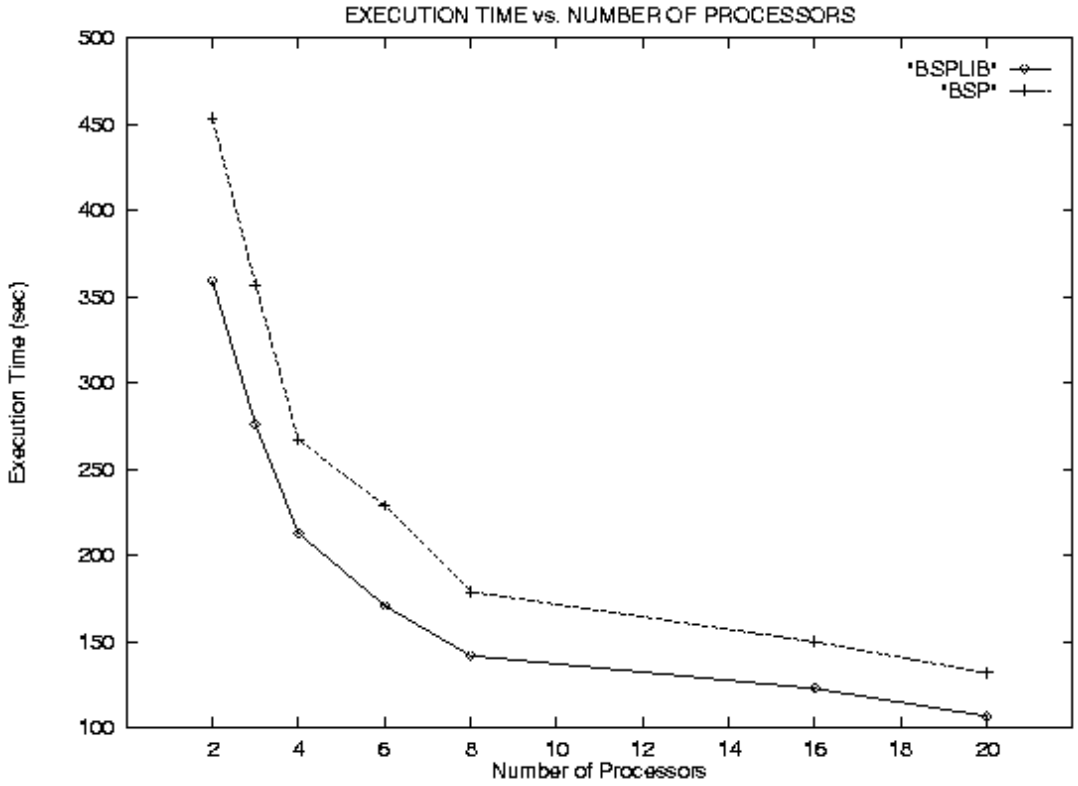

Figure 6.5: Performance comparison of the latest version of BSPLIB(v1.4) and the Oxford BSP Library

The source code for the main module and the communication routines (Gale.cc and BSP_datacomm.cc) of the dedicated network of workstations implementation of GALE using the latest version of BSPLIB (v1.4) is given in Appendix C. The complete source code can be found at the web-site http://www.cs.rpi.edu/kosart/gale-BSP/BSPLIB/ .

7. CONCLUSIONS AND FUTURE WORK

In the previous chapters we investigated the applicability of the BSP model to both dedicated and non-dedicated network of workstations using the application GALE.

We implemented GALE on dedicated network of workstations using the Oxford BSP Library. The results show that BSP based parallel programming can be applied on network of workstations efficiently in spite of the slow network connection and high communication latency.

Then we applied the Adaptive Replication Scheme, developed by M. Nibhanupudi and B. K. Szymanski at Rensselaer Polytechnic Institute [16], to GALE on non-dedicated network of workstations, showing that the use of idle processor cycles in computations gains us almost 1/3 computational power of a dedicated NOW just for zero cost.

Finally, we implemented GALE on dedicated network of workstations using the latest version of BSPLIB and compared its performance with the old version. The results show that the new version introduces a performance improvement of about 27%.

7.1. Future Work

The network used in this project was a 10 Mbps Ethernet, which is considered one of the slowest ones in today's networking technology. Slow network connection means high communication latency, which in term decreases the performance of parallel computations on NOW drastically. If we improved the network speed by using a 100 MB Ethernet or an ATM Network, the performance of both the dedicated and non-dedicated NOWS would increase by a great ratio.

Another improvement on non-dedicated NOW can be done by implementing the adaptive replication scheme using the new version of BSPLIB. The estimated performance improvements on the Execution Time and Speedup are shown in figures 7.1 and 7.2 respectively.

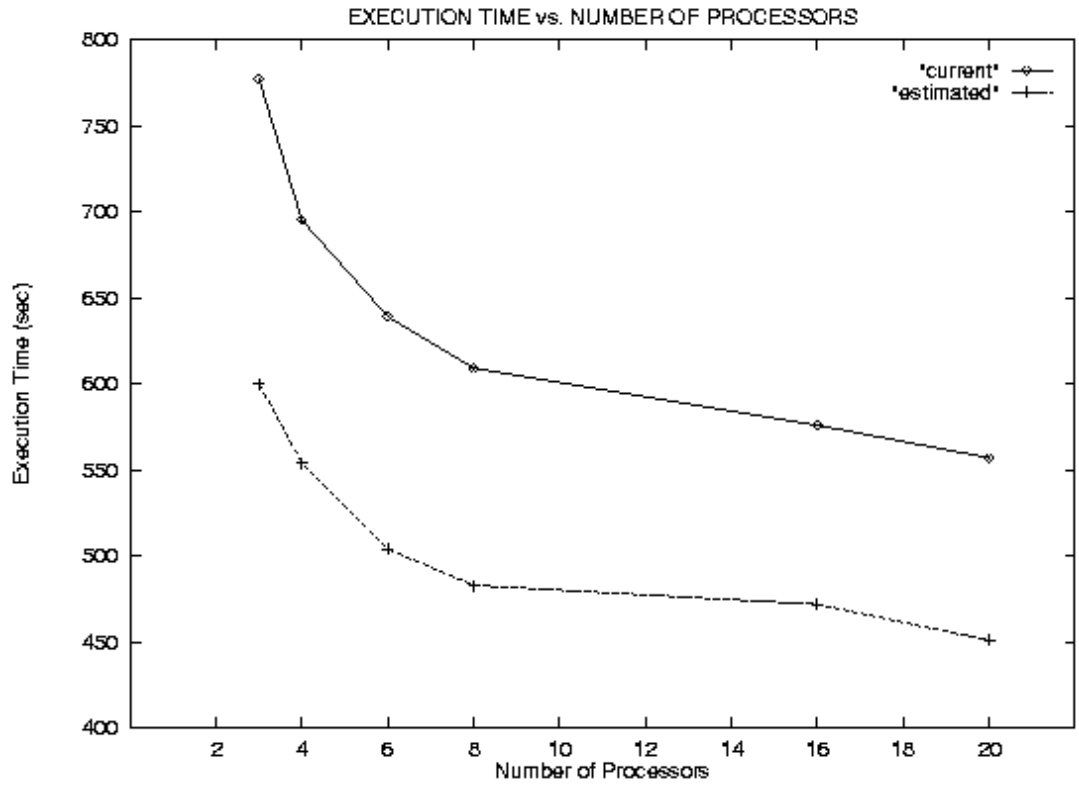

Figure 7.1: Estimated improvements on the Execution Time when the Adaptive Replication Scheme is implemented using the new version of BSPLIB.

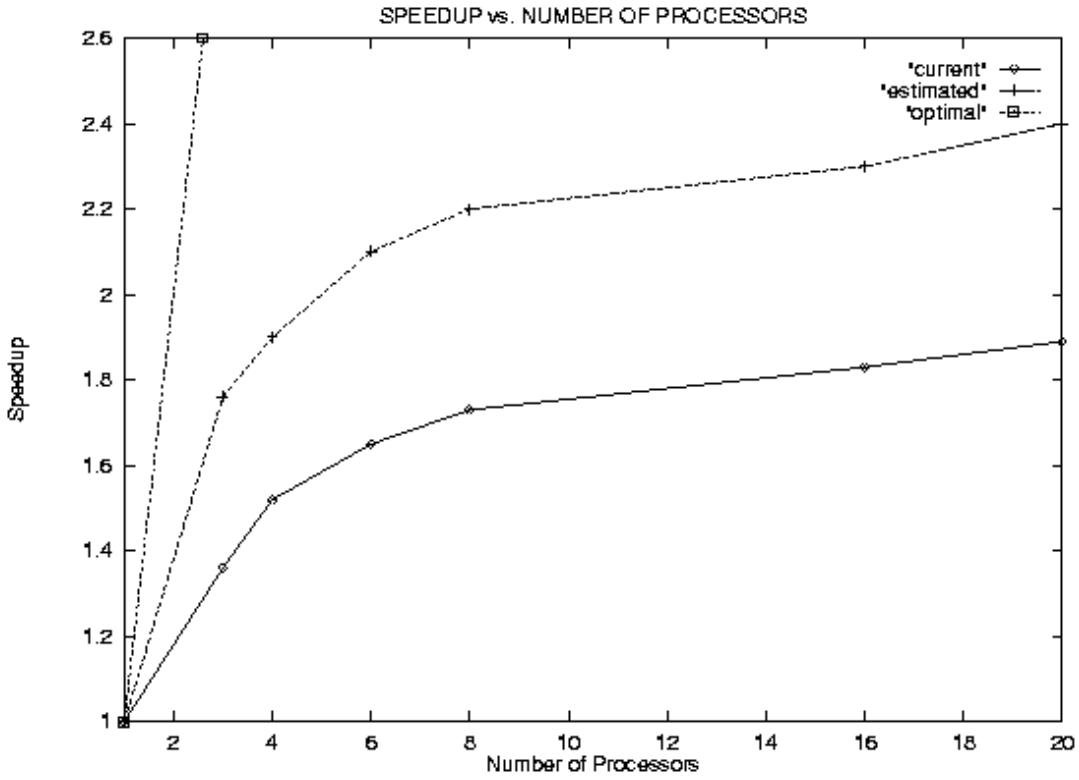

Figure 7.2: Estimated improvements on the Speedup when the Adaptive Replication Scheme is implemented using the new version of BSPLIB.

REFERENCES

[1] T. Anderson and D. Culler, D. Paterson, et. at. “A Case for Network of Workstations (NOW)”. Computer Science Division, EECS Department, University of California, Berkeley, 1994.

[2] M. A. Baker and G. C. Fox. “Distributed Cluster Computing Environments”. Northeast Parallel Architectures Center, January 21, 1996.

[3] S. R. Donaldson, Jonathan M. D. Hill, and D. B. Skillicorn. “ BSP clusters: high performance, reliable and very low cost.” Technical Report PRG-TR-5-98, Programming Research Group, Oxford University Computing Laboratory, September 1998.

[4] Jonathan M. D. Hill. “Collective Communications in the Oxford BSP Toolset”. Oxford University Computing Laboratory, November 27, 1997.

[5] Jonathan M. D. Hill, S. R. Donaldson, A. McEwan. “Installation and User Guide for the Oxford BSP Toolset (v1.4) implementation of BSPlib”. Oxford University Computing Laboratory, September 30, 1998.

[6] Jonathan M. D. Hill, B. McColl, D. C. Stefanescu et. at. “BSPlib – The BSP Programming Library”. May 1999.

[7] R. P. Hudson. “Disease and its Control: The Shaping of Modern Thought”. *Greenwood Press*, Westport, CTT, 1983.

[8] Michael J. Litzkow, M. Livny, and Matt W. Mutka. “Condor - A hunter of idle workstations”. *Eighth International Conference on Distributed Computing Systems*, San Jose, California, June 13-17, 1988, pp. 104-111.

[9] W. Maniatty. “Documentation of Maspar Implementation of TEMPEST Version 3.0”. Rensselaer Polytechnic Institute.

[10] W. Maniatty. “High Performance Computing Tools with Applications to Epidemics and Population Dynamics”. *Ph.D. Thesis*, August 1998.

[11] W. Maniatty, B. Szymanski, T. Caraco. “High-Performance Computing Tools for Modeling Evolution in Epidemics”. January 1999.

[12] W. McColl. “ Bulk Synchronous Parallel Computing *”. 2nd Workshop on Abstract Machines for highly Parallel Computing*, University of Leeds, 14th-16th April 1993.

[13] R. Miller and J. Reed. “The Oxford BSP Library – User’s Guide”. Oxford University Computing Laboratory.

[14] M. V. Nibhanupudi. “Adaptive Parallel Computations on Networks of Workstations”. *Ph.D. Thesis,* Rensselaer Polytechnic Institute, 1998.

[15] M. V. Nibhanupudi , B. K. Szymanski . “ Adaptive Bulk - Synchronous Parallelism in a Network of Nondedicated Workstations”. *Proceedings of the International Symposium on High Performance Computing Systems and Applications*, 1998.

[16] M. V. Nibhanupudi, B. K. Szymanski. “Adaptive Parallelism in the Bu-Synchronous Parallel Model”. *Proceedings of the Second International Euro-Par Conference*, Lyon, France, 1996.

[17] M. V. Nibhanupudi, B. K. Szymanski. “Runtime Support for Virtual BSP Computer”. Lecture Notes in Computer Science, pages 147-158. Springer, 1998.

[18] J. Reed, K. Parrott, T. Lanfear. “Portability, Predictability and Performance for Parallel Computing: BSP in Practice”.

[19] D. B. Skillicorn, Jonathan M. D. Hill and W. F. McColl. “Questions and Answers about BSP”. Programming Research Group, Computing laboratory, Oxford University, November 11, 1996.