

# 16 ADAPTIVE BULK-SYNCHRONOUS PARALLELISM ON A NETWORK OF NON-DEDICATED WORKSTATIONS

Mohan V. Nibhanupudi and Boleslaw K. Szymanski

XXX DEPARTMENT XXX,  
Rensselaer Polytechnic Institute, U.S.A  
{nibhanum, szymansk}@cs.rpi.edu

**Abstract:** XXX CAN WE USE A DIFFERENT TITLE THAT FORMATS BETTER?  
E.G. ADAPTIVE BSP ON A NETWORK OF WORKSTATIONS? XXX Several computing environments, including wide area networks and non-dedicated networks of workstations, are characterized by frequent unavailability of the participating machines. Parallel computations, with interdependencies among their component processes, cannot make progress if some of the participating machines become unavailable during the computation. As a result, to deliver acceptable performance, the set of participating processors must be dynamically adjusted following the changes in computing environment. In this paper, we discuss the design of a run-time system to support a Virtual BSP Computer that allows BSP programmers to treat a network of transient processors as a dedicated network. The Virtual BSP Computer enables parallel applications to remove computations from processors that become unavailable and thereby adapt to the changing computing environment. The run-time system, which we refer to as *adaptive replication system* (ARS), uses replication of data and computations to keep current a mapping of a set of virtual processors to a subset of the available machines. ARS has been implemented and integrated with a message-passing library for the Bulk-Synchronous Parallel (BSP) model. The extended library has been applied to two parallel applications with the aim of using idle machines in a network of workstations (NOW) for parallel computations. We present the performance results of ARS for these applications.

**Keywords:** networks of non-dedicated workstations, bulk-synchronous parallel model, adaptive parallel computing, transient processors, virtual BSP computer.

## 16.1 INTRODUCTION

Several computing environments are characterized by the frequent unavailability of the participating machines. Machines that are available for use only part of the time are referred to as *transient processors* (Kleinrock and Korfhage, 1993). A transition of the host machine from an *available* to a *non-available* state is considered a *transient failure*. Such a model of a network of transient processors applies to several computing paradigms, including wide area networks such as the Internet and local networks of non-dedicated workstations (NOWs). In the latter case, a workstation is available for the parallel computation only when it is *idle*; that is, when it is not being used by its owner; a part of the parallel computation running on a particular workstation must be suspended when its owner activity resumes. Use of workstations in this manner allows additional sequential programs to accumulate work during idle times of the workstations (Kleinrock and Korfhage, 1993). XXX CAN YOU REWRITE THE PREV SENTENCE - IT IS CONFUSING XXX However parallel programs, with interdependencies among their component processes, cannot make progress if some of the participating workstations become unavailable during the computation. Parallel computations in such environments must adapt to the changing computing environment to deliver acceptable performance.

The Bulk-Synchronous Parallel (BSP) model (Valiant, 1990) is a universal abstraction of a parallel computer. By providing an intermediate level of abstraction between hardware and software, BSP offers a model for general-purpose, architecture-independent parallel programming. However the standard libraries for parallel programming using the BSP model offer only static process management (the initial allocation of processors cannot be changed while the parallel computation is in progress) and thus cannot adapt to changing computing environments, such as the ones described above.

In this paper we discuss the design of run-time support for Virtual BSP Computer to enable parallel applications to adapt to the changing computing environment. We refer to the run-time system as the *adaptive replication system* (ARS). We describe our approach to adaptive parallel computations in Section 16.2 and compare it to related work in Section 16.3. Section 16.4 describes adaptive parallelism in the BSP model. In section 16.5 we discuss the design and implementation of the adaptive replication system. The performance of the adaptive replication system is evaluated using two applications in Section 16.6. Finally, we summarize our work and conclude in Section 16.7.

## 16.2 PARALLEL COMPUTING ON NON-DEDICATED WORKSTATIONS

### 16.2.1 Bulk-Synchronous Parallel model

Our view of the parallel computation is based on the Bulk-Synchronous Parallel (BSP) model (Valiant, 1990), in which computation proceeds as a series of *super-steps* comprising of computation and communication operations. All the participating processors synchronize at the end of the super-step. By providing an intermediate level

of abstraction between hardware and software, BSP provides a model for general-purpose, architecture-independent parallel programming. The BSP model has been used to implement a wide variety of scientific applications including numerical algorithms (Bisseling and McColl, 1993), combinatorial algorithms (Gerbessiotis and Siniolakis, 1996) and several other applications (Calinescu, 1995). We used the model to build an efficient implementation of plasma simulation on a network of workstations (Nibhanupudi *et al.*, 1995) as well as a partial differential equation solver for modeling of bioartificial artery (Barocas and Tranquillo, 1994).

Although the barrier synchronization at the end of each BSP super-step can be expensive, its cost can often be reduced by overlapping communication with local computation and enforcing only logical but not physical synchronization. Barriers have desirable features for parallel system design. By making circular data dependencies impossible, they avoid potential deadlocks and live locks, which eliminates the need for their costly detection and recovery (Skillicorn *et al.*, 1997). Barriers ensure that all processes reach a globally consistent state, which allows for novel forms of fault tolerance (Skillicorn *et al.*, 1997). 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 defines a point of a consistent checkpoint. The synchronization at the end of a super-step also provides a convenient point for checking transient process failures. Should one or more processes fail, the surviving processes can start the recovery of the failed processes at this point.

### 16.2.2 Adaptive replication of computations to tolerate transient failures

Our approach relies on executing (replicating) the computations of a failed process on another participating processor to allow the parallel computation to proceed. Note that in the BSP computation, the computation states of the participating processes are consistent with each other at the point of synchronization. By starting with the state of a failed process at the most recent synchronization point and executing its computations on another available participating workstation, we are able to recover the computations of the failed process. This allows the parallel computation to proceed without waiting for the failed process. To enable recovery of the computation of a failed process, the computation state of each process,  $C_s$ , is saved at every synchronization point on a peer process. Thus our approach uses *eager replication of computation state* and *lazy replication of computations*.

## 16.3 RELATED WORK

Piranha (Carriero *et al.*, 1995) is a system for adaptive parallelism built on top of the tuple-space based coordination language *Linda*. Piranha implements master-worker parallelism. The master process is assumed to be persistent. The worker processes are created on idle machines and destroyed when the machine becomes busy. Due to master-worker parallelism, Piranha is applicable to only coarse-grained parallel applications involving independent tasks. Synchronous parallel computations with the computation state distributed among the component processes cannot be modeled with master-worker parallelism.

A limited form of adaptive parallelism can be achieved by dynamically balancing the load on the participating workstations. Parform (Cap and Strumpen, 1993) is a system for providing such capability to parallel applications. Parform is based on the strategy of initial heterogeneous partitioning of the task according to actual loads on workstations, followed by dynamic load balancing. To benefit from Parform, parallel applications must be written in such a way that they can handle a changing communication topology for an arbitrary number of tasks.

There have been several efforts to implement transparent recovery from processor failures for parallel applications. (Leon *et al.*, 1993) discuss the implementation of a consistent checkpointing and roll-back mechanism to transparently recover from individual processor failures. The consistent checkpoint is obtained by forcing a global synchronization before allowing a checkpoint to proceed. CoCheck (Stellner, 1996) tries to blend the resource management capabilities of systems like Condor (Litzkow *et al.*, 1988) with parallel programming libraries such as PVM (Sunderam, 1990) and MPI (Snir *et al.*, 1996; Gropp *et al.*, 1994). It provides consistent checkpointing and process migration mechanism for MPI and PVM applications.

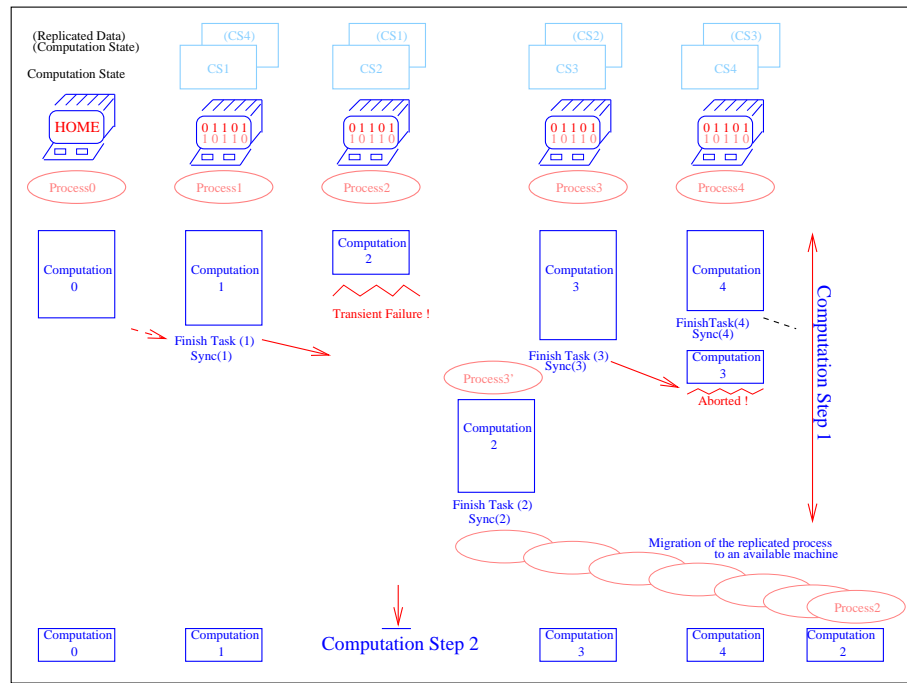
Stardust (Cabillic and Puaut, 1997) is a system for parallel computations on a network of heterogeneous workstations. Stardust uses techniques similar to ours. It captures the state of the computation at the barrier synchronization points in the parallel program. In saving the application state, only the architecture-independent data is saved on disk and transferred to other nodes which allows for the 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 places where an application can be stopped and migrated. Our approach allows for a component process executing on a user's machine to be stopped at any point during the computation. This makes our approach much less intrusive to the individual owners of the workstations and encourages them to contribute their workstations for additional parallel computations during their idle times.

For synchronous parallel applications, our approach provides a less expensive alternative to checkpointing on the disk by replicating the computation state of component processes of the parallel computation on peer processes, which can be considered a form of diskless checkpointing. Our approach offers ways to reduce the amount of data to be replicated on other processes. For example part of the computation state that is common across the component processes need not be replicated on a peer process. As the speed of networks increases relative to the speed of disks, diskless checkpointing should become more attractive for synchronous parallel computations. In addition, replicating computations of a failed process can easily be extended to work across heterogeneous architectures by providing automatic conversion of data representations. Replicating data on peer processes to enable replication of computations in case of failures fits well with emerging architectures like the network computers (OpenGroup, 1997) which may not come with a local disk.

### 16.4 ADAPTIVE PARALLELISM IN BULK-SYNCHRONOUS PARALLEL MODEL

#### 16.4.1 Protocol for replication and recovery

The adaptive replication scheme 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. Figure 16.1 illustrates the protocol. 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. At the beginning of each computation step, 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 computations. Each process also receives the computation state from one or more of its predecessors.



**Figure 16.1** Protocol for replication and recovery illustrated for a replication level of one.

When a process finishes with its computations, 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 using the computation state received from that predecessor. 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. In general, such a newly created process assumes the identity of the predecessor and can continue participating in the parallel computation as a legitimate member. However, for the sake of better performance, this new process is migrated to a new host if one is available. For more details on the protocol, refer to (Nibhanupudi and Szymanski, 1996). It should be noted that the assumption of existence of a master 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.

Systems that intend to provide fault tolerance by transparently recovering from failures rely on some form of replication of resources. In general, we can replicate either data (computation state) or computations or both. When each parallel task is replicated on a group of several processors, it is too costly to update replicas with the status of the fastest processor in each group. On the other hand, migrating a process after discovering the failure intrudes upon the primary user. These considerations led us to the approach in which data are replicated on all processors but the computations are replicated only when needed. In our approach the recovery of the failed computations and subsequent migration to a new available host are performed on an available host, which is much less intrusive.

#### 16.4.2 Tolerating multiple failures

The number of successors at which the computation state of a process is replicated is referred to as the *replication level*, denoted by  $R$ .  $R$  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 consecutive process failures in the logical ring topology that the system can tolerate. Failure of more than  $R$  consecutive processes within the same computation step 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 it also increases the overhead during normal (failure free) execution. The probability of failure of  $R$  consecutive processes is  $P_f^R$ , where  $P_f$  is the probability of failure of a workstation, that is, the probability of a workstation becoming unavailable for computation during a computation step. Since the duration of the computation step is small compared to the mean available and non-available periods, the probability of failure is small ( $P_f \ll 1$ ). Thus the probability of unrecoverable failures decreases exponentially with the replication level  $R$ . The required level of replication to avoid unrecoverable failures is small for the degrees of parallelism used in practice.

### 16.4.3 Performance of adaptive replication

The cost of data replication includes the additional memory required for the replicated data and the cost of transferring the computation state to the successors. The additional memory needed for data replication is proportional to the level of replication,  $R$ , and the size of the computation state,  $C_s$ . The cost of communicating the computation state depends on replication level  $R$ , the size of the computation state  $C_s$  and the underlying communication network. A communication network that scales with the number of processors allows for a higher level of replication and a higher degree of tolerance to transient failures without incurring overhead during normal execution.

To minimize overhead during normal execution, our approach seeks to overlap the computation with communication associated with data replication. For those applications in which the cost of data replication is smaller than the cost of computation in the super-step, replication of computation state can therefore be done without any overhead during normal execution when communication can be overlapped with computation. We refer to such applications as *computation dominant applications*. Under this assumption, the scheme is scalable with high efficiency for small computations, defined as those for which the duration of the computation steps,  $t_s$ , is small compared to the mean length of non-available periods,  $t_n$ . Applications for which the cost of data replication is larger than the computation have an overhead associated with data replication, and therefore they are referred to as *data replication dominant applications*. A more detailed discussion of the performance of the adaptive replication scheme along with the analysis can be found in (Nibhanupudi, 1998).

## 16.5 DESIGN AND IMPLEMENTATION OF THE ARS LIBRARY

XXX YOU ORIGINALLY USED A-BSP IN THE TITLE, BUT NEVER DEFINED IT. YOUR LABEL SAYS ARS, SO I HAVE CHANGED THE FEW OCCURANCES OF A-BSP TO ARS. XXX

### 16.5.1 Design of Adaptive Bulk-Synchronous Parallel Library

The adaptive replication scheme is developed using the Oxford BSP library (Miller, 1993; Miller and Reed, 1993). The library has been extended to provide dynamic process management and virtual synchronization as described in (Nibhanupudi and Szymanski, 1996). Using the extended library, processes can now be terminated at any time or migrated to new host machines; new processes can be created to join the parallel computation. Processes can now perform synchronization for one another, which allows for dynamic work sharing.

The run-time system to support adaptive parallelism in the Bulk-Synchronous Parallel model has been described in detail in (Nibhanupudi and Szymanski, 1998). The run-time support uses two levels of abstraction: *replication layer* and *user layer*. The replication layer implements the functionality of the adaptive replication scheme by providing primitives for replicating the computation state, detecting the failure of a process, replicating the computations of a failed process and restoring the state of a replicated process. This layer is not visible to the user; its functionality can only be accessed through the user layer, which provides the application programming interface.

The user layer includes primitives for specifying the replication data and the computation state of the user process and for performing the memory management required for replication of the computation state. The adaptive extensions in the user layer include constructs to specify computation and communication super-steps. 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. Figure 16.2 shows the constructs provided by the user layer.

```

/* 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 16.2** Adaptive BSP library: user layer.

### 16.5.2 Prototype Implementation

For the prototype we assume that super-steps that make use of replication contain computation only. This is not overly restrictive because, in the BSP model, data communicated in a super-step are guaranteed to be received at the destination process only by the end of the super-step and therefore can only be used in the next super-step. Hence a super-step involving both computation and communication can always be split into a computation super-step and a communication super-step. This assumption greatly simplifies the design of the protocol for the recovery of failed processes. Furthermore, the protocol assumes a reliable network, so a message that is sent by a process will always be received at the destination. The prototype uses a replication level of one.

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. However, using a grace period also delays replicating for the predecessor when required. Based on experimental results, our implementation uses no grace period. 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 new replicated process and allow the predecessor to continue if the predecessor completes its computations before the replicated process or before the super-step is complete.<sup>1</sup> This results in a nice property of the adaptive replication scheme—any processor that is twice as slow as its successor 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. As part of the synchronization, all participating processes receive a list of surviving processes from the master. A participating process that replicated for a predecessor aborts the new replicated process when it finds that the predecessor has successfully performed synchronization.

We are testing our adaptive replication scheme using simulated transient processors with exponential available and non-available periods. A *timer process* maintains the state of the host machine. Transitions of the host machine from an available state to a non-available state and *vice versa* are transmitted to the process via signals. The process is suspended immediately if it is performing a computationally-intensive task such as a computation super-step. Otherwise, the host is marked as unavailable and the process is suspended before entering a computationally-intensive task. The prototype is implemented on Sun Sparcstations using the Solaris operating system (SunOS 5.5). It makes use of the checkpoint-based migration scheme of Condor (Bricker *et al.*, 1992) for process migration.

It should be noted that our adaptive replication scheme protocol can be applied to other message-passing libraries such as MPI (MPI Forum, 1994). The only requirement is that the application be written in the BSP style, using a sequence of computation and communication super-steps.

## 16.6 APPLICATION OF ARS LIBRARY TO PARALLEL COMPUTATION

We applied the ARS library to two different applications that illustrate the performance of the scheme for a *computation dominant* application (maximum independent set) and a *data replication dominant* application (plasma simulation).

### 16.6.1 Maximum Independent set

A set of vertices in a graph is said to be an *independent set* if no two vertices in the set are adjacent (Deo, 1974). A *maximal independent set* is an independent set which 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. 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

---

<sup>1</sup>The super-step is complete when synchronization has been initiated on behalf of all the participating processes.

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. 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 (Goldberg and Hollinger, 1997). To conserve memory, no explicit representation of the graph is maintained. Instead, the connectivity information is used to search through a virtual graph. To reduce the search space, heuristics are used to prune the search space. Each processor searches a subgraph and the processors exchange information on the maximal independent set found on each processor. Since the adjacency matrix is replicated on each processor, the computation state that needs to be communicated to a successor to deal with transient failures is nil. That is, the computation state of a failed process can be recreated based on the knowledge of its identity alone. This application can therefore be categorized as a computation dominant application.

### 16.6.2 Plasma Simulation

The plasma Particle In Cell simulation model (Norton *et al.*, 1995) integrates, in time, the trajectories of millions of charged particles in their self-consistent electro-magnetic fields. In the replicated grid version of the plasma simulation (Nibhanupudi *et al.*, 1995), the particles are evenly distributed among the processors for sharing work load; the simulation space (field grid) is replicated on each of the processors to avoid frequent communication between processors. The computations modify the positions and velocities of the particles, forces at the grid points 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, the forces at the grid points and the grid charge. However, at the beginning of each super-step, all processors have the same global charge distribution and hence the charge data need not be replicated on a remote host. Instead, 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 the discussion in the next section).

Table 16.1(a) 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 (Mutka and Livny, 1987). The measurements were taken on a network of Sun Sparc 5 workstations connected by a 10 Mbps Ethernet. The number of processors available is much larger than the degree of parallelism used in the simulations 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 the execution time on transient processors without using the scheme. Runs on transient processors without the scheme simply suspend the execution of the parallel computation when the host processor is busy. The execution time on a single processor is also shown for reference. As can be seen from these timings, the runs on transient

**Table 16.1** Execution times of (a) maximum independent set problem and (b) plasma simulation on dedicated processors, on transient processors without adaptive replication and on transient processors with adaptive replication. For the runs on transient processors with adaptive replication, the number of migrations during the lifetime of the parallel computation (#moves) is listed in parentheses. All times shown are wall-clock times in seconds.

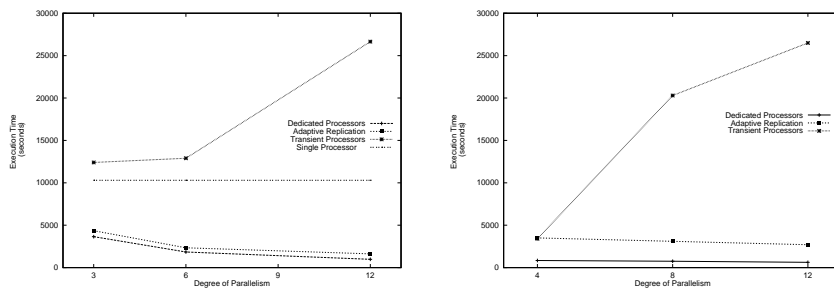
(a) Maximum Independent Set

| Single Proc | Degree of Parallelism | Dedicated Processors | Transient Processors | Transient Processors with Adaptive BSP |              |              |
|-------------|-----------------------|----------------------|----------------------|--|--------------|--------------|
|             |                       | Mean Time            | Mean Time            | Mean (#Trials)                         | Min (#Moves) | Max (#Moves) |
| 10300       | 3                     | 3650                 | 12400                | 4350 (12)                              | 3950 (2)     | 4790 (8)     |
|             | 6                     | 1840                 | 12900                | 2340 (11)                              | 1990 (1)     | 2900 (10)    |
|             | 12                    | 980                  | 26650                | 1620 (9)                               | 1000 (0)     | 2700 (9)     |

(b) Plasma Simulation

| Single Proc  | Degree of Parallelism | Dedicated Processors | Transient Processors | Transient Processors with Adaptive BSP |              |              |
|--------------|-----------------------|----------------------|----------------------|--|--------------|--------------|
|              |                       | Mean Time            | Mean Time            | Mean (#Trials)                         | Min (#Moves) | Max (#Moves) |
| not possible | 4                     | 840                  | 3400                 | 3500 (4)                               | 3340 (3)     | 3774 (6)     |
|              | 8                     | 750                  | 20300                | 3100 (4)                               | 2583 (5)     | 3503 (11)    |
|              | 12                    | 620                  | 26500                | 2700 (3)                               | 2150 (5)     | 3080 (20)    |

processors using the adaptive replication scheme compare favorably with the runs on dedicated processors. Figure 16.3(a) shows a plot of these timings.



**Figure 16.3** Plot showing execution times of (a) maximum independent set and (b) plasma simulation on dedicated processors, on transient processors using adaptive replication and on transient processors without adaptive replication. Execution time on a single processor is shown for comparison purposes.

### 16.6.3 Results

Our measurements indicate that a significant amount of computation was performed using idle workstations. As mentioned in 16.4.1, 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. When using  $p$  processors,  $\frac{p-1}{p}$  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 non-dedicated processors,  $p$  is replaced by the effective number of processors,  $p_{eff} = 1 + (p-1)\frac{t_a}{t_a+t_n}$ , 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,  $p_{eff} = \frac{2p+1}{3}$ . For the dedicated runs, parallel efficiency ranges from nearly 100% (for 3 processors) to 88% (for 12 processors). For the non-dedicated runs using adaptive replication, these values range from nearly 100% (for 3 processors) to 76% at 12 processors. The corresponding values for non-dedicated 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 16.1(b) shows the results of applying the adaptive replication scheme to a plasma simulation with  $N = 3,500,000$  particles. As mentioned in Section 16.6.2, the computation state data that needs to be replicated include the positions and velocities of particles in the local partition and the forces at the grid points in the local partition. The replicated data include 4 floating point numbers for each particle. As a result, for runs with 4 processors, the size of data replicated for particles is about 14 MB. On a 10 Mbps network, replicating the computation state of 3 processors takes up to 40 seconds while the computation step,  $t_s$  is half as long. In addition, the network is shared with other users, so heavy network traffic may increase the time needed for replication. Figure 16.3(b) shows a plot of the execution times 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. Due to the overhead associated with communicating the 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. The execution time on transient processors with adaptive replication is also longer than the sequential execution time, as estimated from the execution times 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. 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 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 replicating the 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 100 Mbps Ethernet, the overhead due to data replication should be much smaller.

## 16.7 CONCLUSIONS

We have proposed an approach to adaptive parallelism based on the Bulk-Synchronous Parallel model to enable parallel computations to tolerate frequent transient failures and thereby adapt to the changing computing environment in a network of non-dedicated workstations. Our approach offers a general framework for adaptive parallelism and is algorithm independent. We described a protocol for replicating the computation state and replicating computations. We extended the Oxford BSP library (Miller, 1993) with dynamic process management and virtual synchronization, and implemented the protocol on top of the extended library. The adaptive parallel extensions to the library include primitives for the specification of replication data, memory management for replication data and specification of computation state. We integrated the adaptive parallel extensions into the Oxford BSP library. The ARS library performs data replication and recovery of failed computations transparently to the user. We have demonstrated the adaptive capabilities of the library by applying it to two applications, a graph search problem and plasma simulation. Our results demonstrate that the ARS library can be used to execute parallel computations efficiently using idle machines in a network of workstations.

### Acknowledgments

This work was partially supported by NSF Grant CCR-9527151. The content does not necessarily reflect the position or policy of the U.S. Government.

### References

- Barocas, V. and Tranquillo, R. (1994). Biphasic theory and in vitro assays of cell-fibril mechanical interactions in tissue-equivalent collagen gels. In Mow, V. *et al.*, editors, *Cell Mechanics and Cellular Engineering*, pages 185–209, New York. Springer-Verlag.
- Bisseling, R. and McColl, W. (1993). Scientific computing on bulk synchronous parallel architectures. Technical Report 836, Department of Mathematics, University of Utrecht.
- Bricker, A., Litzkow, M., and Livny, M. (1992). Condor technical summary. Technical Report CS-TR-92-1069, Computer Science Department, University of Wisconsin, Madison.

- Cabillic, G. and Puaut, I. (1997). Stardust: An environment for parallel programming on networks of heterogeneous workstations. *J. Parallel and Distributed Computing*, 40(1):PAGES XXXXX.
- Calinescu, R. (1995). Conservative discrete-event simulations on bulk synchronous parallel architectures. Technical Report TR-16-95, Oxford University Computing Laboratory.
- Cap, C. and Strumpfen, V. (1993). Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19(11):1221–1234.
- Carriero, N., Freeman, E., Gelernter, D., and Kaminsky, D. (1995). Adaptive parallelism and Piranha. *Computer*, 28(1):40–49.
- Deo, N. (1974). *Graph Theory with Applications to Engineering and Computer Science*. Prentice-Hall, Inc., Englewood Cliffs, N.J.
- Gerbessiotis, A. and Siniolakis, C. (1996). Selection on the bulk synchronous parallel model with applications to priority queues. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, page PAGES XXXXX, Sunnyvale, California, USA.
- Goldberg, M. and Hollinger, D. (1997). Database learning: A method for empirical algorithm design. In *Workshop on Algorithm Engineering*, page PAGES XXXXXX.
- Gropp, W., Lusk, E., and Skjellum, A. (1994). *Using MPI: Portable Parallel Programming With the Message-Passing Interface*. MIT Press.
- Kleinrock, L. and Korfhage, W. (1993). Collecting unused processing capacity: An analysis of transient distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):PAGES XXXXX.
- Leon, J., Fischer, A., and Steenkiste, P. (1993). Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.
- Litzkow, M., Livny, M., and Mutka, M. (1988). Condor - a hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, page PAGES XXXXX, San Jose, California.
- Miller, R. (1993). A library for bulk-synchronous parallel programming. In *British Computer Society Workshop on General Purpose Parallel Computing*, page PAGES XXXXX.
- Miller, R. and Reed, J. (1993). The Oxford BSP library users' guide, version 1.0. Technical report, Oxford Parallel.
- MPI Forum (1994). *MPI: A Message Passing Interface Standard*. Technical report, Message Passing Interface Forum.
- Mutka, M. and Livny, M. (1987). Profiling workstations' available capacity for remote execution. In *12th Symposium on Computer Performance*, page PAGES XXXXX, Brussels, Belgium.
- Nibhanupudi, M. (1998). *Adaptive Parallel Computations on Networks of Workstations*. PhD thesis, Computer Science Department, Rensselaer Polytechnic Institute.
- Nibhanupudi, M., Norton, C., and Szymanski, B. (1995). Plasma simulation on networks of workstations using the bulk-synchronous parallel model. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'95)*, page PAGES XXXXX, Athens, Georgia.

- Nibhanupudi, M. and Szymanski, B. (1996). Adaptive parallelism in the bulk-synchronous parallel model. In *2nd International Euro-Par Conference*, page PAGES XXXXX.
- Nibhanupudi, M. and Szymanski, B. (1998). Runtime support for virtual BSP computer. To appear in the Workshop on Runtime Systems for Parallel Programming, RTSP'98.
- Norton, C., Szymanski, B., and Decyk, V. (1995). Object oriented parallel computation for plasma PIC simulation. *Communications of the ACM*, 38(10):PAGES XXXXX.
- OpenGroup (1997). *Network Computer Profile*. Technical report, The Open Group.
- Skillicorn, D., Hill, J., and McColl, W. (1997). Questions and answers about BSP. *Scientific Programming*, 6(3):249–274.
- Snir, M., Otto, S., Huss-Lederman, S., and Walker, D. (1996). *MPI: The Complete Reference*. Scientific and Engg Computation Series. MIT Press.
- Stellner, G. (1996). CoCheck: Checkpointing and process migration for MPI. In *International Parallel Processing Symposium*, page PAGES XXXX.
- Sunderam, V. (1990). PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339.
- Valiant, L. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.