High Performance Cluster Computing, vol. I, Architectures and Systems,
Prentice Hall, New York, 1999, pp. 702-721

Contents

Chapter 1

BSP-Based Adaptive Parallel Processing

Mohan Nibhanupudi and Boleslaw Szymanski

Rensselaer Polytechnic Institute, Troy, New York
Email: {*nibhanum, szymansk*} *@cs.rpi.edu*

1.1 Introduction

In this chapter, we focus on clusters consisting of a group of workstations connected through a local area network, often run under a single administration. In particular, we target clusters with fast communication network achieved thanks to low-overhead protocols and use of switched networks that allow bandwidth to scale with the number of processors. Message passing libraries such as PVM [17], MPI [10] and BSP Oxford Library [9] allow for portable parallel programs. The SPMD (Single Program Multiple Data) paradigm lends the programmer flexibility in structuring parallel applications with varying degrees of granularity. Accordingly, we explore parallel processing on clusters of nondedicated workstations using the Bulk-Synchronous Parallel model. We extend the BSP model to enable the BSP computation to adapt to the changing degree of parallelism available on clusters of nondedicated workstations and demonstrate its use for efficient parallel programming.

1.2 The Bulk-Synchronous Parallel Model

The Bulk-Synchronous Parallel model [18] defines an abstract parallel computer in terms of the following primitives:

- *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 communicates with other components. The data communicated is 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 as 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 parameters: number of processors (p), processor speed (s), synchronization periodicity (L), and a parameter to indicate the global computation to communication balance (g). The processor speed is measured in the number of time steps executed 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 are dependent on the number of processors p. This dependency is defined by the network architecture and the implementation of the communication and synchronization primitives.

For example, consider a cluster of workstations interconnected by a fixed bandwidth communication medium such as an Ethernet. Communicating large amounts of data 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 follows: $g(p) = g_0 p$, where g_0 is a constant. If we assume that the synchronization mechanism is implemented in software using a tree structure for the participating processors, the parameter L is defined as $L = L_0 \log(p)$, where L_0 is 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 [5] is to charge $\max\{l, w + gh_s, w + gh_r\}$ time steps for the superstep. Yet another definition [2] 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 asymptotic costs of a BSP superstep computed according to the above formulae are of the same complexity. The cost of the entire

BSP algorithm is just the sum of the costs of its supersteps.

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 [2].

1.2.1 Cluster 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 cluster 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 of local area networks in comparison with the custom-design switching networks of parallel architectures). As a result clusters of workstations may not efficiently execute algorithms designed for parallel computers. For example, to mask high value of g, every non-local memory access should 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 organizing the participating processors into a (binary) tree 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 communicates the data to the processors at its child nodes. 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 a number of steps proportional to the logarithm of the number of processors involved. The cost of logarithmic broadcast of h units of data on a cluster is

$$L\log(p) + g(p-1)h = L_0 \log^2(p) + g_0(p-1)h \tag{1.2.1}$$

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

$$L + g(p-1)h = L_0 \log(p) + g_0(p-1)h \tag{1.2.2}$$

Comparing 1.2.1 and 1.2.2 shows that, unlike in a parallel computer environment, linear broadcast is always faster in a cluster 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.

1.2.2 Program Reorganization for Parallel Computing on Dedicated Clusters: Plasma Simulation

The Particle-in-Cell (PIC) method simulates the trajectories of millions of particles in their self-induced fields. The interactions between the particles are modeled indirectly through the fields induced by the particles at the fixed points of a grid. The General Concurrent Particle-in-Cell (GCPIC) algorithm partitions the particles and grid points uniformly among the processors of a distributed memory machine. This allows for efficient computation of positions and velocities of the particles. As particles move among partitioned regions, they are passed to the processor responsible for the new region. To enable efficient solution of the field equations on the grid, 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 is 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 is replicated on the neighboring processor to avoid frequent off-processor references.

The distributed grid described above require a parallel machine with fast interconnection because interactions between particles and grid points belonging to different processors gives rise to frequent communication. To improve performance of plasma simulation on a cluster, we use a replicated grid that eliminates communication associated with interactions of particles on one processor with grid points on another. It also eliminates communication associated with solving the field equations on a distributed grid. In addition, a replicated grid allows particles to remain on the same processor for the entire duration of the simulation eliminating communication associated with particle redistribution.

As a result, the replicated grid version of plasma simulation performs well on clusters of workstations [12]. This application demonstrates that it is possible to execute computation intensive parallel applications on a cluster of workstations. However, the application may need to be restructured by changing the data distribution to avoid frequent communication.

1.3 Parallel Computing on Nondedicated Workstations

Workstations in a cluster are often under-utilized [11], [3]. Arpaci et al. [1] 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 in a cluster to run additional parallel jobs.

1.3.1 Nondedicated Workstations as Transient Processors

There have been several systems that attempt to make use of idle workstations to execute sequential programs [8]. Such additional computation is suspended when primary user activity is detected to avoid performance degradation for primary users. It is resumed when primary user activity ends and the workstation becomes idle. The workstations that are available for use only when they are idle are referred to as *transient processors* [7]. A transition of the host processor from an available to a non-available 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 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. [7] who showed that the rate of progress 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 non-available period, making use of parallelism inefficient. In some cases synchronous parallel programs may take longer to execute on nondedicated clusters of workstations than on a single workstation sequentially. To deliver acceptable performance, parallel applications executing in such environments must be able to adapt to the changing computing environment; we refer to such ability as *adaptive parallelism*.

1.3.2 Approaches to Adaptive Parallelism

Recall that a transition of the host processor from an available to a non-available state is 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 non-available to an available state is treated as *recovery* of this 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 a system: prevent (or avoid) occurrence of failures or recover from them. Prevention or avoidance of failures is usually achieved through redundancy, i.e., use of multiple instances of certain critical resources. In case of parallel computations, we can use multiple instances of either data or computations to prevent or avoid failures. Alternatively, we can try to recover after a failure has occurred by re-executing the failed (delayed) computations.

Based on these general principles, we identify three schemes to deal with transient failures. The schemes 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 the replication takes place, as in Table 1.1. The straightforward execution simply delays the completion of the computation step until all participating processes finish their computation, even if some of the processors participating in the computation change their state from available to non-available. This scheme requires the least effort, but is also susceptible to the full impact of the unavailability of participating processors.

Table 1.1 Classification of Schemes to Deal with Transient Processor Failures Using Replication of Data and Computations as well as Migration of Processes

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

The first approach to reduce the impact of transient failures is based on eager (preventive) replication of component processes which increases the probability that at least one replica finishes the computation step without transient failure. Such replication can be justified by the argument that the idle time on a processor is free for use and, therefore, costs nothing. This approach is called the *full replication scheme*. 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 problem with this solution is that it is often too costly to update replicas with the status of the fastest processor in each group.

Another approach is based on *recovery from failures* by another component process executing on a different processor than the failing one. We further assume that at least one of the component processes is immune to transient failures due to processor unavailability. 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.

Yet another approach is to deal with transient failures preventively. 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. Recovery of computations in this approach is distributed among the components and hence this scheme has the potential to be scalable. Due to the advantages this scheme offers, we choose this scheme to implement adaptive parallelism in the Bulk-Synchronous Parallel model.

1.4 Adaptive Parallelism in the Bulk-Synchronous Parallel Model

As explained above, 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 the Bulk-Synchronous Parallel 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. Thus, our approach uses *eager replication of computation state and lazy replication of computations.*

1.4.1 Protocol for Replication and Recovery

The master process coordinates recovery from transient failures without replicating for any of the failed processes. Figure 1.1 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.

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 — and restores the computation state of each new process 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 [13]. 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

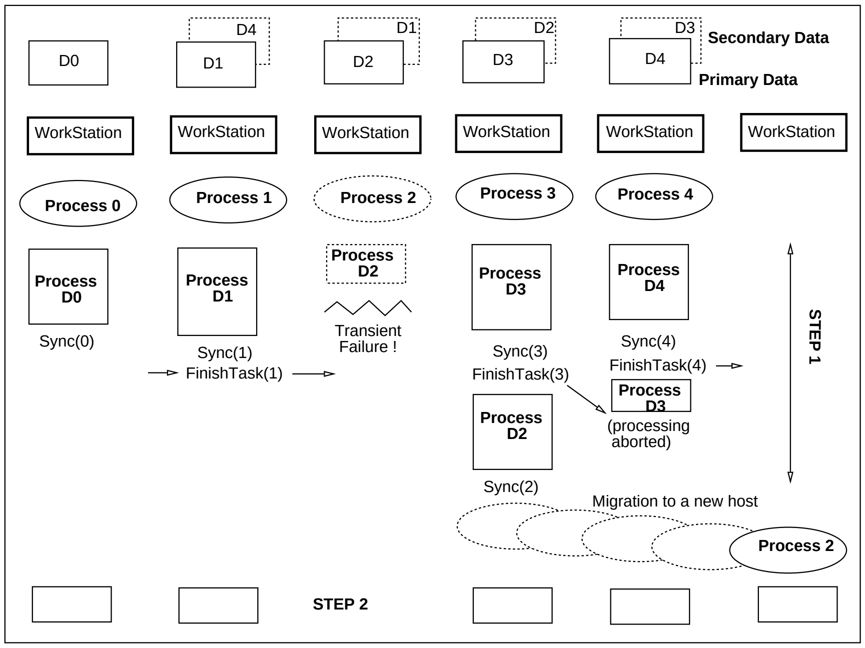


Figure 1.1 Protocol for replication and recovery illustrated for a replication level of one.

prototypical implementation of the system.

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 than migrating from the failed (i.e., non-available) process.

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 until 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 transient failure of R consecutive processes is P_f^R, where P_f is the probability of transient failure of a single workstation. Assuming the duration of the computation step is small compared to the mean available and non-available periods, the probability of failure is small ($P_f \ll 1$). This assumption is justified because we are interested in the small total parallel computation time, so the larger the computation, the more processors we are willing to use. Hence, the computation step on each processor is relatively small, regardless of the size of the application. Under this assumption, the probability of irrecoverable failures decreases exponentially with the replication level R. The optimal for scalability level of replication growths as a logarithm of the number of processors and for sufficiently large degrees of parallelism (that are most important for practical applications) is very small.

1.4.2 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 the 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 overlaps 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 superstep, replication of the computation state can be done without any overhead during normal execution. We refer to such applications as *computation dominant applications*. Under these assumptions, the scheme is scalable with high efficiency. 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 [15]. It should be noted that for scalability, it is sufficient that the cost of data replication will be of the same order (when expressed as a function of the problem size and the number of processors) as the cost of computation in a superstep.

1.5 A Programming Environment for Adaptive Bulk-Synchronous Parallelism

A programming environment developed at Rensselaer known as Adaptive BSP (A-BSP) library, is designed within the framework of the BSP model [18] and developed using the Oxford BSP Library [9]. A-BSP consists of dynamic extensions to the Oxford BSP library and the adaptive replication scheme designed in two levels of abstraction: *replication layer* and *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. These primitives are accessible only through the user layer. By designing the run-time 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.

1.5.1 Dynamic Extensions to the Oxford BSP Library

The Oxford BSP Library implements a simplified version of the Bulk-Synchronous Parallel model. It is simple, yet robust, and was successfully used by us for implementing plasma simulation on a cluster of workstations [12]. We extended the Oxford BSP Library to provide dynamic process management and virtual synchronization as described in [13]. 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.

The A-BSP prototype implementation was based on the following assumptions. The supersteps that make use of adaptive replication contain computation only. This is not overly-restrictive, since a superstep containing computation and communication can always be expressed as a sequence of computation and communication supersteps. This assumption greatly simplifies the design of the protocol for the recovery of failed processes. We assume a reliable network, so a message that is sent by a process will always be received at the destination.

In A-BSP, restoring the computation state of the failed process involves

- restoring specific system state from the backup copy received from that process,
- restoring common system state from local checkpoint, and
- executing the user supplied recovery function.

Each of the newly-created processes performs the computations on behalf of a failed process and performs synchronization on its behalf to complete the computation step. In general, such a newly created process assumes the identity of the corresponding failed process and can continue participating in the parallel computation as a legitimate member. However, for the sake of better performance, this restored process is migrated to a new host if one is available.

1.5.2 The 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 is created as 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 common 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.

1.5.3 The User Layer

The user layer provides the application programming interface (API) for the A-BSP library. It includes the following primitives that transparently allow access to the functionality of the replication layer:

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

  The construct `bsp_replication_data` (see Figure 1.2 for the full syntax) allows the user to specify data to be replicated. The user can specify static storage for replication data by defining a valid location for the `store` parameter. Otherwise, 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 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 system state can be specified for a computation superstep using the construct `bsp_specific_system_state` and the common system state using the construct `bsp_common_system_state`.

```
/* Constructs to specify a computation superstep */
bsp_comp_sstep(int sstepid);
bsp_comp_sstep_end(int sstepid);
/* 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);
bsp_specific_system_state(BspSystemState* bss);
bsp_common_system_state(BspSystemState* bss);
RecoveryFunction();
```

Figure 1.2 Adaptive parallel extensions to the Oxford BSP Library (User Layer).

- 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 1.2 - 1.5 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 1.2 shows the constructs provided by the user layer described above. Figure 1.3 illustrates the use of these constructs to specify replication data. Figure 1.4 illustrates the use of the constructs to specify the computation state of a component process. Figure 1.5 illustrates the use of the A-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 to the user.

```
/* 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): A 2 dimensional array, with no static storage available
             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 1.3 Use of A-BSP constructs to specify replication data.

1.6 Application of A-BSP to Parallel Computations

We applied the A-BSP library to two different applications that illustrate the performance of the scheme for *computation dominant* applications and *data replication dominant* applications described in Section 1.4.2.

```
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 1.4 Use of A-BSP constructs to specify computation state.

1.6.1 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 [4]. 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. Given 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

```
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 1.5 An A-BSP computation superstep.

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 [6]. 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.

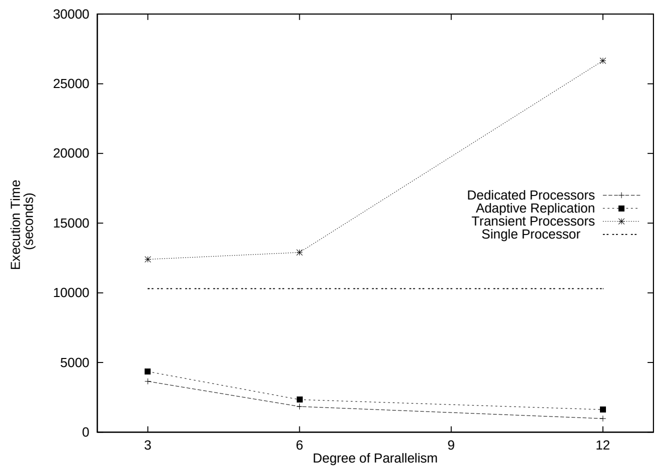
1.6.2 Plasma Simulation

The plasma Particle-in-Cell simulation model was described in Section 1.2.2. In the replicated grid version of this model [12], the particles are evenly distributed among the processors 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 needs to be replicated includes the positions and velocities of the particles, the forces at the grid points and the grid charge. However, at the beginning of each superstep, all processors have the same global charge distribution and hence the charge data does not need to 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 discussion in Section 1.6.3). It is still scalable, as long as the interconnection network of an executing cluster is scalable.

1.6.3 Results

Figure 1.6(a) shows a plot of the execution times of maximum independent set problem on transient processors using the A-BSP library 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 [11]. The measurements were taken on a cluster 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 A-BSP library were compared with the execution time on dedicated processors and with execution time on transient processors without using the adaptive replication scheme. Runs on transient processors that do not use A-BSP 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 processors using the A-BSP library compare favorably with runs on dedicated processors. Our measurements indicate that a significant amount of computation was performed using idle workstations. Since a dedicated workstation is used to execute the main process, when using a parallelism of p, a fraction of $\frac{p-1}{p}$ of the total computation is performed by the idle machines.

Figure 1.6(b) shows the results of application of A-BSP library to plasma simulation with $N = 3,500,000$ particles. As mentioned in Section 1.6.2, the computation state data that needs to be replicated includes the positions and velocities of particles in the local partition and the forces at the grid points in the local partition. The replicated data includes four floating point numbers for each particle. As a result, for runs with four processors, the size of data replicated for particles is about 14 MBytes. On a 10 Mbps cluster, replicating the computation state of three processors takes up to about 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 1.6(b) shows a plot of 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. For plasma simulation, due to the overhead associated with 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. 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, the 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



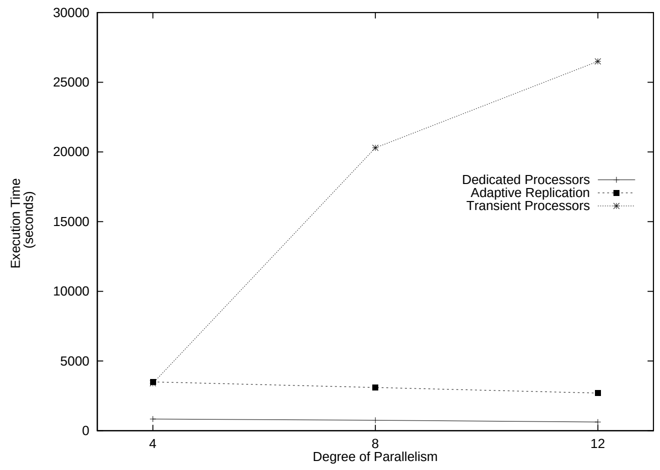


Figure 1.6 Plot showing execution times of (a) maximum independent set and (b) plasma simulation on dedicated processors, on transient processors using adaptive replication and on transient processors without adaptive replication. Execution time on a single processor is shown for comparison purposes.

not available for parallel computation, application of A-BSP library 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 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 100 Mbps Ethernet, the overhead due to data replication should be much smaller.

1.7 Application of A-BSP to Clusters of Nondedicated Workstations

The results shown in Section 1.6 are obtained with simulated transient processors with exponentially distributed available and non-available periods. In this section, we present the results of executing a graph search algorithm using the adaptive replication on a cluster of nondedicated workstations in the Department of Computer Science at Rensselaer.

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

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. 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. 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.

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 replicated on a backup process. The replicated adjacency information is used to recover the predecessor in case of a failure. Replication of the adjacency matrix partition can be done at the beginning of the search. When a process performs the recovery for its failed predecessor and assumes its identity, the adjacency matrix information is no longer valid since the process now has a different predecessor. The new process needs to update the adjacency information from its predecessor. 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.

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.

The processor pool used for these runs consisted of about 20 machines that included Sparc 5 (Models 110 and 70) and Sparc 20 processors. A host monitor [14] is used to determine the status of the workstations based on the cpu load and the activity of the console user. The runs used a degree of parallelism of 6, with 5 of them using nondedicated machines. 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 scalable 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.

1.8 Conclusions

In this chapter, we described a programming environment for clusters of nondedicated workstations to facilitate efficient parallel computations. Our approach to adaptive parallelism is based on the Bulk-Synchronous Parallel model. It enables parallel computations executing on nondedicated workstations to tolerate frequent unavailability of the workstations in a nondedicated cluster and thereby adapt to the changing computing environment. Our approach offers a general framework for adaptive parallelism and is application independent. We described a protocol for the replication of computation state and replication of computations. We extended the Oxford BSP library [9] 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 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 A-BSP 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 A-BSP library can be used to execute parallel computations efficiently using idle machines in a cluster of nondedicated workstations.

1.9 Bibliography

[1] 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.

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

[3] Clemens H. Cap and Volker Strumpen. Efficient Parallel Computing in Distributed Workstation Environments. *Parallel Computing*, vol. 23, pages 1221–1234, 1993.

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

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

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

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

[8] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, California, June 13-17, 1988.

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

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

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

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

[13] M. V. Nibhanupudi and B. K. Szymanski. Adaptive Parallelism in the Bulk-Synchronous Parallel Model. In *Proceedings of the 2nd International Euro-Par Conference*, Lyon, France, August 1996.

[14] M. V. Nibhanupudi and B. K. Szymanski. Adaptive Parallel Computing on Nondedicated Networks of Workstations Using the Bulk Synchronous Parallel Model. *Technical report*, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, April 1998.

[15] Mohan V. Nibhanupudi. *Adaptive Parallel Computations on Networks of Workstations.* Ph.D. Thesis, Computer Sciences Department, Rensselaer Polytechnic Institute, 1998.

[16] D. A. Nichols. Using Idle Workstations In A Shared Computing Environment. In *Proceedings of the 11th ACM Symposium on Operating System Principles*, ACM, November 1987.

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

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

Index