Runtime Support for Virtual BSP Computer [⋆]

Mohan V. Nibhanupudi and Boleslaw K. Szymanski

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY, USA 12180-3590
{nibhanum, szymansk}@cs.rpi.edu

Abstract. Several computing environments including wide area networks and nondedicated networks of workstations are characterized by frequent unavailability of the participating machines. Parallel computations, with interdependencies among their component processes, can not 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.

1 Introduction

Several computing environments are characterized by frequent unavailability of the participating machines. Machines that are available for use only part of the time are referred to as *transient processors* [5]. A transition of the host machine from an *available* to a *non-available* state is considered a *transient failure*. Such model of a network of transient processors applies to several computing paradigms, including wide area networks such as the Internet and local networks of nondedicated 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 parallel computation running on a particular workstation must be suspended when its owner activity resumes. Use of

[⋆] 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.

workstations in this manner allows additional sequential programs to accumulate work during idle times of the workstations [5]. However parallel programs, with interdependencies among their component processes, can not 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.

Bulk-Synchronous Parallel (BSP) model [14] 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 2 and compare it to related work in section 3. In sections 4 and 5, we discuss the design and implementation of the adaptive replication system. In section 6, we briefly discuss the performance of the adaptive replication system and present performance results for two applications. Finally, we summarize our work and conclude in section 7.

2 Adaptive Parallel Computations in Virtual BSP Computer

2.1 Model of Parallel Computation

In the Bulk-Synchronous Parallel model [14] the parallel computation proceeds as a series of *supersteps* comprising of computation and communication operations. All the participating processors synchronize at the end of the superstep. 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 superstep which eliminates the need for consistent checkpointing. The synchronization at the end of a superstep also provides a convenient point for checking process failures. Should some processes fail, surviving processes can start recovery of the failed processes at this point.

2.2 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. By starting with the state of a failed process at the most recent synchronization point and executing its computations, we are able to recreate the state of the failed process. This allows the parallel computation to proceed without waiting for the failed process. This approach takes into account the nature of the computing environment in nondedicated NOWs in which the machine cycles are relatively inexpensive since we are mainly using idle machine cycles.

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

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

3 Related Work

Piranha [3] is a system for adaptive parallelism based on the tuple-space based coordination language *Linda*. Piranha implements master-worker parallelism and hence 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 [2] is a system for providing such capability to parallel applications.

Leon et. al. [6] discuss 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 [12] tries to blend the resource management capabilities of systems like Condor [7] with parallel programming libraries such as PVM [13] and MPI [4]. It provides consistent checkpointing and process migration mechanism for MPI and PVM applications. Stardust [1] is a system for parallel computations on a network of heterogeneous workstations. It captures the state of the computation at the barrier synchronization points in the parallel program. A major limitation of Stardust's mechanism of using naturally occurring synchronization barriers is that it limits the number of points where an application can be stopped and migrated.

Our approach allows for a component process executing on a user's machine to be suspended 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. In addition, our approach to replicating computations of a failed process can easily be extended to work across heterogeneous architectures by providing automatic conversion of data representations.

4 Design of the Adaptive Replication System

The Adaptive Replication System is designed within the framework of the BSP model [14] and developed using the Oxford BSP Library [8]. ARS consists of dynamic extensions to the Oxford BSP library and the adaptive replication scheme. The adaptive replication scheme is designed in two levels of abstraction: *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. However, these primitives are not directly accessible to the applications; the functionality provided by the replication layer can be accessed only through the user layer. By designing the runtime support in two layers, we intend to insulate the applications from changes in the implementation. By implementing the replication layer for other architectures, we can maintain the portability of applications using our library.

4.1 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 network of workstations [9]. We extended the Oxford BSP Library to provide dynamic process management and virtual synchronization as described in [10]. 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.

4.2 Protocol for Replication and Recovery

For the prototype under implementation we make 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. We further assume 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.

The participating processes other than the master process are organized into a logical ring topology in which each process has a predecessor and a successor. Each process in the ring communicates its specific system state to one or more of its successors, called *backup processes*, before starting its own computations, where it is stored as a backup copy (SSS-BACKUP). Each process also saves the common system state as a local checkpoint (CSS-BACKUP). When a process finishes with its computations, it sends a message indicating successful completion to 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 it holds. 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 each computation step. Restoring the computation state of the failed process involves

- restoring specific system state from the backup copy received from that process (SSS-BACKUP),
- restoring common system state from local checkpoint (CSS-BACKUP), 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.

4.3 Adaptive Replication Scheme: Replication Layer

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

- Replicate the specific system state on the backup process as determined by the replication protocol.

– Checkpoint the common system state locally on the same process.
– Detect the failure of the process whose computation state is replicated on this process.
– Create a new process to execute the computations of a failed process. The new process 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 local system states.
– Execute the recovery function supplied by the user.
– Perform synchronization on behalf of a failed process.
– Terminate lagging processes whose computations have been successfully replicated.
– Migrate the process to another available host.

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

4.4 Adaptive Replication Scheme: User Layer

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

– Constructs to specify data to be replicated and to specify memory management for the replication data.
  The construct `bsp_replication_data` (see Figure 1 for the full syntax) allows the user to specify data to be replicated. The user can specify static storage for replication data by specifying a valid location for the `store` parameter. When no storage is explicitly specified by the user (by passing a `0` value), automatic memory management is assumed and the system allocates dynamic storage for the replication data. It keeps track of the dynamic storage across process replications.
– Constructs to specify computation state.
  A predefined structure `BspSystemState` can be used to declare variables that hold 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.
  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 - 4 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 shows the constructs provided by the user layer described above. Figure 2 illustrates the use of these constructs to specify replication data. Figure 3 illustrates the use of the constructs to specify the computation state of a component process. Figure 4 illustrates the use of the extended BSP construct for the computation superstep. The specific and local system states must be specified for each computation superstep. The computation superstep requires no additional constructs; adaptive replication and recovery of failed computations is done transparently.

```
/* 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();
```

Fig. 1. Adaptive parallel extensions to the Oxford BSP Library (User Layer)

5 Implementation of the Adaptive Replication System

We have implemented the adaptive replication system as additional layers on top of the Oxford BSP library. The ARS is available as a library of C functions and

```
/* 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);
```

Fig. 2. Use of extended-BSP constructs to specify replication data

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

Fig. 3. Use of extended-BSP constructs to specify computation state

can be used by parallel applications in the same way a BSP library is used. In implementing the prototype, we have assumed a replication level[2] of one. That is, a process can act as a backup for its immediate predecessor only. The prototype is implemented on Sun Sparcstations using the Solaris (SunOS 5.5) operating system. It makes use of the checkpoint based migration scheme of Condor [7] for process migration. It should be noted that our protocol for adaptive replication scheme can be applied to other message passing libraries such as MPI [4].

[2] Replication level is the number of processes on which the computation state of a process is replicated. It defines the maximum number of successive (transient) processor failures (according to the order of the processes in the logical ring topology, see section 4.2) that the adaptive replication system can tolerate. Refer to [10] for more details.

```
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 );
```

Fig. 4. A BSP computation superstep with adaptive replication.

The only requirement is that the application be written in the BSP-style, as a sequence of computation and communication supersteps.

5.1 Failure Detection and Replication of Computations

In the adaptive replication scheme, a process starts replicating for its predecessor when it concludes that its predecessor has failed. Failure detection is 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. Due to the coarse grain nature of the applications, gang scheduling [11] is not required. 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. Our implementation allows the user to specify the grace period. However, based on experimental results, we have not used a grace period with the applications that we tested. A process starts replicating for its predecessor if it has not received a message of successful completion from the predecessor by the time it finishes its own computations. However, to avoid unnecessary migrations, we abort the replicated process and allow the predecessor to continue if the predecessor finishes its computations before the replicated process or before the synchronization is complete. This results in a nice property of the adaptive replication scheme - any processor that is twice as slow as its successor and slower than all other processes 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 currently available machines.

5.2 Coordination of Distributed Events

The adaptive replication system uses a combination of signals, messages and locking to coordinate and control events that occur at the component processes. Following is a list of events that are handled by ARS.

– Completion of the computation by a component process is communicated to the successor by a message indicating successful completion.
– Migration of a process to another available host is achieved by checkpointing the process and restarting the process from the checkpoint on the target host. The process can restart only after the checkpointing is complete. Coordination of these events is handled through file locking.
– A process delayed due to a transient failure of its host and whose computations have been successfully replicated needs to be terminated. Termination of a lagging process is done by sending the process a `SHUTDOWN` message.
– A newly created process replicating for a delayed process needs to be terminated if the delayed process manages to finish its computations before the new process. Termination of the replicated process is done by the parent process which created the new process.

6 Performance of Adaptive Replication System

The cost of data replication includes the additional memory required for the replicated data and the cost of communicating the computation state to the backup processes. To minimize overhead during normal execution, our approach seeks to overlap communication associated with data replication with the computation. Overhead incurred by data replication depends on the characteristics of the application as well as the characteristics of the communication medium such as the network bandwidth, latency of communication protocols, ability of the network to overlap communication with computation, etc. Applications in which the data replication can be done without a significant delay to computation are referred to as *computation dominant applications.* Applications in which the data replication incurs a significant overhead are referred to as *data replication dominant applications.* 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 nonavailable 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 superstep. Otherwise, the host is marked as unavailable and the process is suspended before entering a computationally intensive task.

The Adaptive replication system has been applied to two different applications that illustrate the performance of the scheme for *computation dominant* applications and *data replication dominant* applications. Figure 5 shows the performance of the adaptive replication system for computation dominant and replication dominant application respectively. For computation dominant application the performance of ARS is comparable to that of dedicated processors. In the maximum independent set problem, there is no data to be replicated and the overhead of adaptive runs is due to replication of failed computations and migration of processes. For replication dominant applications, the adaptive replication system incurs the overhead of replication of computation state. In plasma simulation, data replication accounts for about 50% of the execution time of the adaptive runs and recovery of failed computations accounts for 25%. Even in this case, execution time of the parallel application when using the adaptive replication is significantly less than the execution time on transient processors without using adaptive replication. For large computations that do not fit on a single machine, adaptive replication ensures that the parallel runs using transient processors complete in a reasonable time. These measurements were obtained using Sun Sparc 5 machines connected by a relatively slow 10 Mbits/s Ethernet. With a faster communication network, the overhead due to data replication will be smaller and the performance of data replication dominant applications will be correspondingly higher.

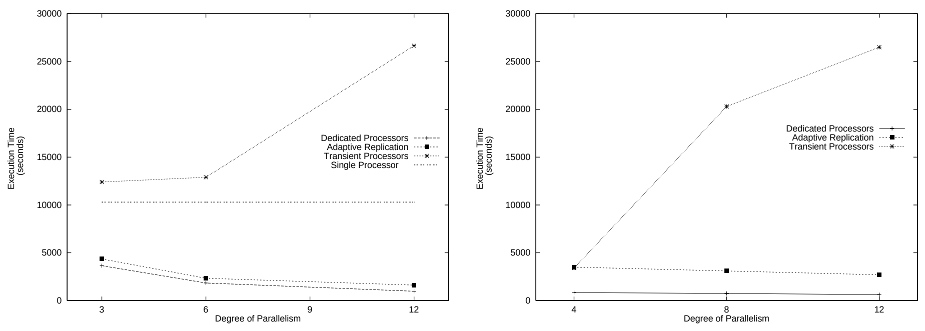

Fig. 5. Plot illustrating the performance of adaptive replication system for (a) computation dominant application (maximum independent set problem) and (b) replication dominant application (plasma simulation). The plot shows the execution times on transient processors using adaptive replication, on transient processors without adaptive replication and on dedicated processors. Execution time on a single processor is shown for comparison purposes.

7 Conclusions

We have designed a run time system to support a Virtual BSP Computer that allows a BSP programmer to treat a network of transient processors as a dedicated network. The runtime system is based on an adaptive parallelism approach using eager replication of computation state and replication of computations.

The adaptive replication system is applicable to parallel applications written in BSP-style and is developed as a set of layers on top of the Oxford BSP library. By separating the implementation of the adaptive replication scheme from the application programming interface, our design allows for extensibility and portability. The Virtual BSP Computer can be ported to other message passing libraries such as MPI by reimplementing the replication layer, while providing the same interface to application programmers. The run time system has been successfully applied to two different parallel applications using idle machines in a network of workstations.

References

1. Gilbert Cabillic and Isabelle Puaut. Stardust: an environment for parallel programming on networks of heterogeneous workstations. *J. Parallel and Distributed Computing*, 40(1), Jan 1997.
2. Clemens H. Cap and Volker Strumpen. Efficient Parallel Computing in Distributed Workstation Environments. *Parallel Computing*, pages 1221–1234, 1993.
3. Nicholas Carriero, Eric Freeman, Gelernter, and David Kaminsky. Adaptive Parallelism and Piranha. *Computer*, 28(1):40–49, January 1995.
4. Message Passing Interface Forum. MPI: A Message Passing Interface Standard. Technical report, Message Passing Interface Forum, May 5, 1994.
5. L. Kleinrock and W.Korfhage. Collecting Unused Processing Capacity: An Analysis of Transient Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(5), May 1993.
6. J. Leon, Allan L. Fischer, and Peter Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical report, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, Feb 1993.
7. Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. 8th Intl. Conf. Distributed Computing Systems*, San Jose, California, June 13-17, 1988.
8. Richard Miller. A Library for Bulk-synchronous Parallel Programming. In *British Computer Society Workshop on General Purpose Parallel Computing*, Dec 1993.
9. M. V. Nibhanupudi, C. D. Norton, and B. K. Szymanski. Plasma Simulation On Networks Of Workstations Using The Bulk-Synchronous Parallel Model. In *Proc. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'95)*, Athens, Georgia, Nov 1995.
10. M. V. Nibhanupudi and B. K. Szymanski. Adaptive Parallelism In The Bulk-Synchronous Parallel model. In *Proceedings of the Second International Euro-Par Conference*, Lyon, France, Aug 1996.
11. J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proc. Third Intl. Conf. Distributed Computing Systems*, Oct 1982.
12. G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the International Parallel Processing Symposium*, April 1996.
13. V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.
14. Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

This article was processed using the LaTeX macro package with LLNCS style