Proc. EurPar96 Parallel Processing, Lyon, France, August 1996, vol. II, LNCS 1124,
Springer Verlag, Berlin, 1996, pp. 311-318

Adaptive Parallelism in the Bulk-Synchronous Parallel Model *

Mohan V. Nibhanupudi and Boleslaw K. Szymanski

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY, USA 12180-3590

Abstract. The Bulk-Synchronous Parallel (BSP) model is a universal abstraction of parallel computation that can be used to design portable parallel software. Advances in processor architecture and network communication enable clusters of workstations to be used as parallel computers. This paper focuses on using the idle computing power of a network of workstations to run parallel programs. The transient nature of the processors causes straightforward execution of synchronous BSP programs to perform poorly in such an environment. In this paper, we propose a scheme, based on the eager replication of state data and lazy replication of processes, that allows BSP programs to run efficiently on transient processors. The scheme is integrated into the Oxford BSP library.

Keywords: BSP Model, Networks of Workstations, Adaptive Parallel Computing

1 Introduction

The Bulk-Synchronous Parallel (BSP) model [12] is a universal abstraction of parallel computation which can be used to design portable parallel software. Advances in micro processor architecture and data communication allow distributed memory machines and networked workstations to deliver high performance for parallel applications. The focus of this paper is on a network of workstations viewed as a BSP computer. However, unlike the majority of the research on high performance distributed computing, we are interested in utilizing idle computing power of the workstations without disrupting services for regular users (owners) of the machines. The problem is challenging for highly synchronous applications because the performance is limited by the progress of the slowest or least available machine. Another complication is the unpredictability of machine load, i.e. each currently free workstation can suddenly become unavailable for a long time.

There have been several systems that attempt to make use of idle time on workstations. Typically (see [2]), such systems avoid degrading the performance for the owner of the machine by requiring that the background computation be

* This work was partially supported by NSF Grants CCR-9216053 and CCR-9527151. The content does not necessarily reflect the position or policy of the U.S. Government.

suspended when owner's activity is detected. The background computation is resumed when the owner's activity ends and the processor is idle. Since these processors are available for use only when they are idle and not available at other times, they are referred to as "transient" processors [4]. Consequently, we treat a change of the workstation status from free to unavailable as a transient failure. Another assumption of our approach is that the frequency of synchronization of parallel computation is high compared with the average available/nonavailable times of a workstation.

In this paper, we propose extensions to the Oxford BSP Library [6, 7] that allow for transparent recovery from transient processor failures in a network of workstations using dynamic replication of computation state, lazy process replication and process migration. The resulting system supports parallel computation with the level of parallelism changing in response to the changes in the number of available workstations. We refer to this kind of computation as *adaptively parallel.*

2 Parallel Computation on a Network of Transient Processors using the BSP Model

In the BSP model the computation consists of a sequence of supersteps. In each superstep, the processors perform some local computation and initiate communication to other processors. All the processors synchronize at the end of each superstep. Consequently, if a participating processor becomes unavailable while executing a superstep, the progress of the entire program will be stalled.

In [4], Kleinrock et. al. found the probability density of a program's finishing time on multiple processors in which duration of available and nonavailable periods are independent and identically distributed random variables from a general distribution assuming long duration programs (that execute for several available periods) composed of many independent tasks. Hence, each processor can proceed independently of the others. As a result, the average processing time of the program with execution time T on a dedicated workstation is equal to $\frac{T}{p} * \frac{t_a+t_n}{t_a}$ on p nondedicated workstations, where t_a, t_n are average available and nonavailable times.

The impact of transient failures on tightly synchronized parallel programs with relatively small amount of computation between synchronizations is much more severe. In [10], Nibhanupudi and Szymanski analyze the execution time of a BSP program with relatively short supersteps on a network of transient processors for the special case of exponentially distributed available and nonavailable times. Many scientific applications such as the plasma simulation [9] that require frequent communication and synchronization among the component processes belong to this category. The results of the analysis in [10] are summarized below. Consider a BSP computation with the average inter-synchronization times measured on a single dedicated workstation equal to $T \ll t_a$ (and also $T \ll t_n$). Since all the processors synchronize at the end of the computation, the finishing time of the step is the maximum of the finishing times of the computation on

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

The delay caused by nonavailable workstations can be decreased by replicating the component processes on more than one processor, thereby increasing the chances of timely delivery of the results of each process. Replication of the component processes is expensive because the computations done by all but one of the replicated processes are discarded. Instead, in this paper we take the approach of lazy (i.e. invoked when needed) replication of the computations. That is, when a transient processor fails (becomes unavailable), then its computation is performed by an available processor and computations are replicated only when required by the failure of a processor. This approach saves computation but sequentializes failure recovery because the processor running a replica must finish its regular computation first.

Our approach to adaptive parallelism is based on viewing the unavailability of a host machine as a transient failure; the effect of the transient failure is to delay the parallel computation. Our approach deals with transient failures through *data replication*[3] of computation state and *time redundancy*[2] [8] of computations. In distributed systems, data objects are replicated on multiple nodes to increase the availability of the data and thereby increase the resiliency to failures. Time redundancy is used to deal with the effects of temporary failures. In our approach, replication of computation state is done in order to allow for repetition of computations of a failed process, should a host processor become unavailable. This view of a transient failure includes in its scope any real failure of the host machine. Thus our scheme can recover from real failures of host machines that occur after the data replication has completed successfully. However, we are mainly concerned about the adaptability of the parallel application to the changing computing environment to maintain acceptable performance on a nondedicated network of workstations. To provide fault-tolerance to hardware and software failures, the presented mechanism would have to be supplemented with a recovery system based on periodic checkpointing of the component processes.

2.1 Adaptive Parallelism through Replication and Migration

In our scheme for lazy replication of computations, each process is ready to act as a backup for one or more of its peers. This is achieved by eager replication of the computation state of a component process on one or more of its peers. In the event of the failure of a component process, one of the backup processes can take over and complete the computations of the failed process. This is further discussed in section 3. The superstep is complete when all the

[2] *Time redundancy* refers to the repetition of a computation or communication action in the domain of time.

computations are successfully completed by either the respective component processes or their back-ups. However, for subsequent supersteps, it is desirable to remove the failed process from the parallel computation and avoid replication which creates load imbalance between processors. A scheme to exploit adaptive parallelism, therefore, should provide for migrating processes across machines. This approach requires no additional resources, since no processes are replicated and replication is lazy in that computations are repeated only when a processor has failed or is delayed. The only investment is data replication, main cost of which is the communication needed to send the state of each subcomputation to peer processes.

2.2 Extensions to the Oxford BSP Library

The above described scheme of using lazy replication and migration to support adaptive parallelism has been integrated into the Oxford BSP library [6, 7]. It implements a simplified version of the Bulk-Synchronous Parallel model [12] introduced by Leslie Valiant. It is simple, yet robust and was successfully used by us for implementing plasma simulation on a network of workstations.

The extensions to the Oxford BSP library are introduced in two layers. The first layer implements our lazy replication and migration scheme. This layer is integrated into the BSP library. The second layer implements a small set of primitives that can be used by the programmer to specify the data structures that constitute the computation state.

The extensions in the first layer support the following new features.

1. The processes are dynamic: processes can be terminated; new processes can be created and enter the computation; processes can migrate from one processor to another.
2. At synchronization one process can be substituted by another so dynamic work sharing among the component processes is possible.
3. The extended library implements our protocol for lazy replication of computations and migration of the failed process to an available processor.

3 Lazy Replication and Migration in the BSP model

Lazy replication alleviates the impact of transient processor failures (unavailability) by ensuring that computation will proceed in spite of the failure. The tolerance of the system to transient failures depends on the degree of replication. We refer to the degree of replication as *replication level* denoted by R. A system with a replication level of R can survive the failure of R successive component processes. Lazy replication is implemented as follows. The participating processes, except process 0, are organized in to a logical ring, thereby establishing a total order between themselves. Process 0 is assumed to run on a host owned by the user running adaptive parallel computation which makes it immune to transient failures. At the beginning of every superstep, each process

communicates the state of its computations in the current superstep to R of its successors, where R is a desired reliability level, discussed in the following paragraph. When the process finishes its computations, it sends a completion message to its R successors. It then checks if it has received completion messages from its R predecessors. After a short timeout period, lack of such a message is interpreted as the failure of the predecessor. The process then creates a new process for each failed predecessors. Such newly created process first restores the state of the failed process, then executes its computations and finally performs synchronization on its behalf. In general, such 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 renewed process is migrated to a new host if one is available.

We refer to the data to be processed (data comprising the computation state) as the *primary data* and its replica at a successor process as the *secondary data*. Figure 1 illustrates the working of the replication scheme for a replication level of one.

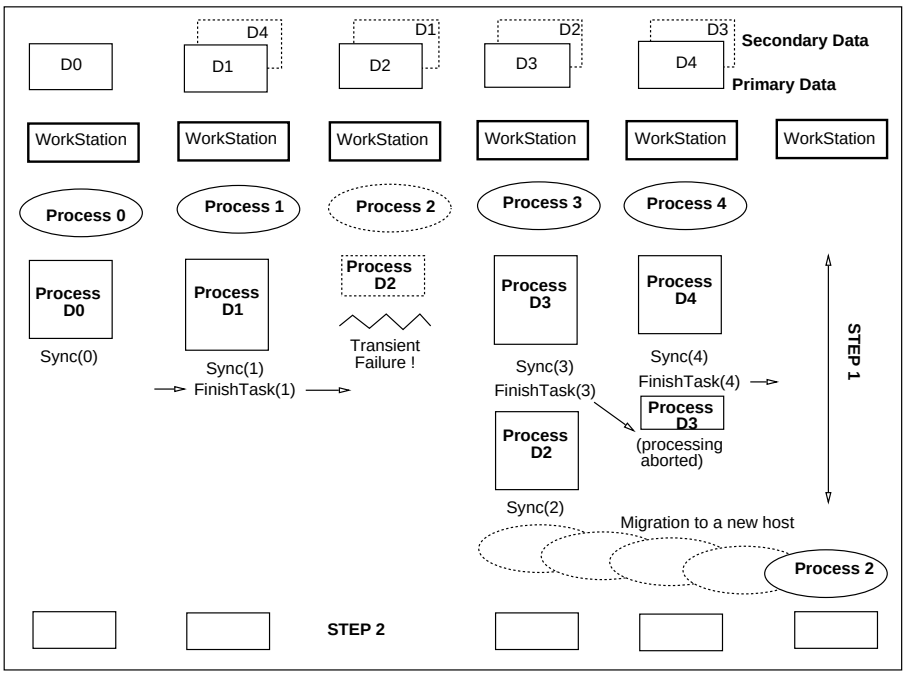

Fig. 1. The Replication Scheme for a replication level of 1.

Our lazy replication scheme is based on the following assumptions. There will be no node failures during the initial stage of the superstep when the process communicates its state to the successors. This is a reasonable assumption,

because we require that the computation will not be interrupted by the owner's processes only during the period when communication is initiated, which is quite short. Additionally, we assume a reliable network, so there will be no network failures; a message that has been sent by a node will be received at the destination node. Further, we assume (for the prototype under implementation) that supersteps that make use of replication contain computation only. This is not overly restrictive, because in the BSP model, data communicated in a superstep are guaranteed to be received at the receiving process only at the end of the superstep and can only be used in the next superstep. Hence a superstep involving both computation and communication can always be split into a computation superstep and a communication superstep. We intend to relax this restriction in the future versions.

The BSP approach provides a very general framework[5] and most applications can be represented as BSP computations. Our scheme for adaptive parallelism in the BSP approach is based on replicating the computation state and repeating the computation step in the event of failure of the component process. Hence the scheme is applicable to computation supersteps that transform the computation state of the process without any side effects. Since the failed computations are repeated by another process on a possibly different processor, we require that the computation superstep not include operations with side effects such as output, memory allocation, etc.

4 Plasma Simulation

We have implemented the lazy replication scheme in the plasma simulation [9]. The plasma Particle In Cell simulation model [1] integrates in time the trajectories of millions of charged particles in their self-consistent electromagnetic fields. Particles can be located anywhere in the spatial domain; however, the field quantities are calculated on a fixed grid. In the General Concurrent Particle in Cell (GCPIC) Algorithm [11], both the particles and the field are partitioned among the processors. In the replicated grid version of the plasma simulation [9], the simulation space is replicated on each of the processors. The particles are evenly distributed among processors in the primary decomposition, which makes advancing particle positions in space and computing their velocities efficient. Replicating the simulation space (grid) on each processor avoids frequent communication between processors otherwise needed because of interactions between particles on one processor with a portion of the grid on another processor.

5 Implementation and Results

We are implementing the described scheme for adaptive parallelism in layers. The first layer consists of a set of primitives implemented in the Oxford BSP library. The replication scheme builds upon this layer and is embedded into the *bspsstep* and *bspsstep_end* calls that delimit a superstep. This layer implements

the replication scheme transparently to the user. However, the replication scheme requires knowledge of which data structures constitute the process state for any superstep. The proposed second layer implements the primitives which the user can use to specify these structures. We are thinking of ways to extract this information from the user program unobtrusively.

We have currently implemented a prototype that implements our lazy replication scheme. For the sake of quick implementation, the prototype is embedded in the user program. We have also made some simplifying assumptions. For example, in the prototype we have assumed a replication level of one. As mentioned in section 3, our prototype requires that a superstep making use of replication contain computation only. Further, we have currently implemented a simple scheme to choose a host machine to which the new process will migrate using the migration scheme of Condor [2] for this purpose. Apart from the use as a proof of concept, this implementation will help us in understanding the issues that arise when the replication scheme is available in a BSP library and in refining the set of needed primitives.

We are testing our replication and migration scheme using simulated transient processors with exponential available and non-available periods. We have used mean available and mean nonavailable periods of 30 minutes and 20 minutes respectively. Our preliminary results were obtained using a plasma simulation with 300,000 particles and a level of parallelism of 4. For simulations using our adaptive replication and migration scheme, we assumed an infinite pool of processors, so that an *available* processor, to which a failed process can be migrated, is always available and therefore the initial level of parallelism can be maintained throughout. The execution time on a single dedicated processor is 5400 seconds under light load conditions and 9200 seconds under heavy load conditions. The execution time using our replication and migration scheme using idle machines (equally distributed during light and heavy load conditions) ranged from 5100 seconds during light load conditions to 9400 seconds during heavy load conditions, with an average of 6400 seconds. The execution time of the parallel simulation without using our scheme is significantly larger. Our measurements indicate that a significant amount of computation was performed using idle workstations. The minimum execution time using the scheme is less than the minimum execution time using a single dedicated processor by about 5%. We expect the performance of the scheme to improve for large problem sizes using a higher level of parallelism. For significantly large problems that do not fit into the main memory of a single processor, one will be required to use parallelism. We intend to measure the performance of the replication scheme for various values of such parameters as the number of processors, mean values of the available and nonavailable periods, etc.

6 Conclusion

We have proposed and implemented a lazy replication and migration scheme that allows a parallel program running on a network of transient processors to

recover from the transient failure (unavailability) of processors. We have proposed extensions to the Oxford BSP Library to implement the scheme. The scheme is lazy since replication of computation is done only when required. The only investment is that of sending the state of the computation to a peer process. The proposed BSP library extensions implement the scheme transparently to the BSP user. We are exploring ways to extract the state information from the user through the use of annotations.

References

1. C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation.* The Adam Hilger Series on Plasma Physics. Adam Hilger, New York, 1991.
2. A. Bricker, M. Litzkow, and M. Livny. Condor Technical Summary. Technical Report CS-TR-92-1069, Computer Sciences Department, University of Wisconsin-Madison, Jan 1992.
3. Pankaj Jalote. *Fault Tolerance in Distributed Systems.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1994.
4. L. Kleinrock and W.Korfhage. Collecting Unused Processing Capacity: An Analysis of Transient Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):535–546, May 1993.
5. W F McColl. BSP Programming. In G Blelloch, M Chandy, and S Jagannathan, editors, *Proc. DIMACS Workshop on Specification of Parallel Algorithms*, Princeton, May 94. American Mathematical Society.
6. Richard Miller. A Library for Bulk-synchronous Parallel Programming. In *British Computer Society Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, December 1993.
7. Richard Miller and Joy Reed. The Oxford BSP Library Users' Guide, version 1.0. Technical report, Oxford Parallel, 1993.
8. Sape Mullender. *Distributed Systems.* ACM Press Frontier Series. ACM Press, New York, 2nd edition, 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 *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'95)*, pages 13–22, Athens, Georgia, November 3-4, 1995.
10. M. V. Nibhanupudi and B. K. Szymanski. Efficiency Of Parallel Computation Replication On A Network Of Transient Processors. Submitted to Eighth IEEE Symposium on Parallel and Distributed Processing to be held in October 1996.
11. C. D. Norton, B. K. Szymanski, and V. K. Decyk. Object Oriented Parallel Computation for Plasma PIC Simulation. *Communications of the ACM, 38(10)*, October 1995.
12. 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