Process Migration and Fault Tolerance of *BSPlib* Programs Running on Networks of Workstations

Jonathan M.D. Hill[1], Stephen R. Donaldson[1] and Tim Lanfear[2]

[1] Oxford University Computing Laboratory, UK.
[2] British Aerospace Sowerby Research Centre, UK.

Abstract. This paper describes a system that enables parallel programs written using the *BSPlib* communications library to migrate processes among a network of workstations. Not only does the system provide fault tolerance of *BSPlib* jobs, but by utilising a load manager that maintains an approximation of the global load of the system, it is possible to continually schedule the migration of BSP processes onto the least loaded machines in a network. Results are provided for an industrial electromagnetics application that show that we can achieve similar throughput on a publically available collection of workstations as a dedicated NOW.

1 Introduction

The Bulk Synchronous Parallel (BSP) model [14, 10] views a parallel machine as a set of processor-memory pairs, with a global communication network and a mechanism for synchronising all processors. A BSP program consists of a sequence of *supersteps*. Each superstep involves all of the processors and consists of three phases: (1) processor-memory pairs perform a number of computations on data held locally at the start of a superstep; (2) processors communicate data into other processor's memories; and (3) all processors barrier synchronise. The globally consistent state that is available after the barrier not only helps when reasoning about parallel programs, but also suggests a programming discipline in which computation (and communication) is balanced across all the processes. As balance is so important to BSP, profiling tools have concentrated upon exposing imbalances to the programmer so that they can be eliminated [5, 7]. However, not all imbalances that arise during program execution are caused by the program. In an environment where processors are not dedicated resources, the BSP computation proceeds at the speed of the slowest processor. This would suggest that the synchronous nature of BSP is a disadvantage compared to the more lax synchronisation regime of message passing systems such as MPI. However, most programs written using collective communications, or scientific applications such as the NAS parallel benchmarks [1] are highly synchronous in nature, and are therefore limited to the performance of the slowest running process in either BSP or MPI. Therefore, if a network user logs onto a machine that is part of a BSP job, this may have an undue effect on the entire job. This paper describes a technique that ensures a p process BSP job continually adapts itself to run on the p least loaded processors in a network consisting of P machines ($p \leq P$).

Dedicated parallel machines can impose a global scheduling policy upon their user community such that, for example, parallel jobs do not interfere with each other in a detrimental manner. The environment that we describe is one where it is not possible to impose some schedule on the user community. The components of the parallel computation are invariably guests on other peoples machines and should not impose any restrictions on them for hosting the computation. Further, precisely because of this arrangement, the availability of the nodes and the available resources at these nodes is quite erratic and unpredictable. We adopt the philosophy that in such a situation it is reasonable to expect the parallel job to look after itself.

We briefly describe the steps involved in migrating a BSP job, that has been written using the *BSPlib* [6] communications library, among a set of machines and the strategy used in making check-pointing and migration decisions across all machines. The first technical challenge (Section 2) describes how we capture the state of a UNIX process and restart it in the same state on another machine of the same type and operating system. The simplicity of the superstep structure of BSP programs provides a convenient point at which local checkpoints capture the global state of the entire BSP computation. This therefore enables process migration and check-pointing to be achieved without any changes to the users program. Next we describe a strategy whereby all processes simultaneously decide that a different set of machines would provide a better service (Section 3). When the BSP job decides that processes should be migrated, all processes perform a global checkpoint, they are then terminated and restarted on the least loaded machines from that checkpoint. Section 4 describes a technique for determining the global load of a system, and Section 5 describes an experiment using an industrial electro-magnetic application on a network of workstations. This demonstrates how the scientist or engineer is allowed to concentrate on the application and not on maintaining or worrying about the choice of processors in the network. Section 6 describes some related work and Section 7 concludes the paper.

2 Check-Pointing and Restarting Single Processes

BSPlib provides a simple API for inter-processor communication in the context of the BSP model. This simple interface has been implemented on four classes of machine: (1) distributed memory machines where the implementation uses either proprietary message passing libraries or MPI; (2) Distributed memory machines where the implementation uses primitive one sided communication, for example the Cray SHMEM library of the T3E; (3) shared memory multi-processors where the implementation uses either proprietary concurrency primitives or System V semaphores; and (4) Networks of workstations where the implementation uses TCP/IP or UDP/IP. In this paper we concentrate upon check-pointing programs running on the network of workstations version of the library. Unlike other check-pointing schemes for message passing systems (See Section 6), by choosing to checkpoint at the barrier synchronisation that delimits supersteps, because there

is a globally consistent state upon exiting the barrier (where all communication is quiesced), the task of performing a global checkpoint reduces to the task of check-pointing all the processes at the local process level.

All that is required to perform a local checkpoint is to save all program data that is active. Unfortunately, because data (i.e., the state) can be arbitrarily dispersed amongst the stack, heap and program text, capturing the state of a running program is not as simple as it would first appear. A relatively straightforward solution in a UNIX environment is to capture an image of the running process and create an executable which contains the state of the modified data section (including any allocated heap storage) and a copy of the stack. When the check-pointed executable is restarted the original context is restored and all *BSPlib* supporting IPC connections (pipes and sockets) are re-established before the computation is allowed to proceed. All this activity is transparent to the programmer as it is performed as part of the *BSPlib* primitives. Furthermore, by restricting the program to the semantics of *BSPlib*, no program changes are required. The process of taking a checkpoint involves making a copy of the stack on the heap, saving the current stack pointer and frame pointer registers, and saving any additional state information (for example, on the SPARC the register windows need to be flushed onto the stack before it is saved, and the subroutine return address needs to be saved as it is stored in a register; in contrast, on the Intel X86 architecture, all necessary information is already stored on the stack). The executable that captures this state information is built using the `unexec()` function which is distributed as part of Emacs [11]. The use of `unexec()` is similar to its use (or the use of `undump`) in Emacs, LaTeX (which build executables containing initialised data structures) and Condor which also performs check-pointing [4]. However, the state saving in Condor captures the point of execution and the stack height using the standard C functions `setjmp()` and `longjmp()` which only guarantee far jumping into activation records already on the stack and within the same process instance. Instead, we capture the additional required information based on the concrete semantics of the processor. To restart a process and restore its context, the restart routine adjusts the stack pointer to create enough space on the stack so that the saved stack can be copied to its original address and restores any saved registers.

3 Determining when to Migrate Processes

As mentioned above, our philosophy is that the executing job be sensitive to the environment in which it is executing and it is the job, and not an external scheduler, that makes appropriate scheduling decisions. For the job to make an informed decision, some global resource information needs to be maintained. The solution we have adopted is that there are daemons running on each machine in the network which maintain local approximations to the global load. The accuracy of these approximations is discussed in the next section. Here we assume that each machine contains information on the entire network which is no more than a few minutes old with a high probability.

When a BSP job requests a number of processes, the local daemon is queried for a suitable list of machines on which to execute the job (the daemon responds so that the job may be run on the least loaded machines). In order that the decisions are not too fickle, the five minute load averages are used. Also, it is a requirement that not too much network traffic be generated to maintain a reasonable global state. Since the five minute load averages are being used, it is not too important that entries in the load table become slightly out of date as the wildest swings in the load averages take a couple of minutes to register in the five minute load average figures.

Let G_i be the approximation to the global load on machine i, then given P machines, the true global load is $G = G_1 \sqcup \cdots \sqcup G_P$; where $\sqcup$ is used to merge the approximations from two machines. Given a BSP job running on p processors with machine names[1] j in the set $\{i_1, \ldots, i_p\}$, we use the approximation $G' = G_{i_1} \sqcup \cdots \sqcup G_{i_p}$ which is better than any of the individual approximations with a high probability. G' is a sequence of machine names sorted in decreasing priority order (based on load averages, number of CPUs and their speeds). If the top set of p entries of G' is not $\{i_1, \ldots, i_p\}$ then an alternate and better assignment of processes to machines exists (call this predicate *fn*). In order not to cause processes to thrash between machines, a measure x of the load of the job (where $0 \leq x \leq 1$) is added to the load averages of all machines not involved in the BSP computation before the predicate *fn* is applied. This anticipates the maximum increase in the load of a machine when a process is migrated to it. Any observed increase in load greater than x is therefore caused by additional external load.

Our aim is to ensure that the only overhead in process migration is the time taken to write p instances of the check-pointed program to disk. Therefore, we require that the calculation that determines when to perform process migration does not unduly impact the computation or communication performance of *BSPlib*. We need to solve $fn(G') = fn(G_{i_1} \sqcup \cdots \sqcup G_{i_p})$ either on a superstep by superstep basis or every N supersteps. However, the result can be obtained without first merging the global load approximations. This can be done by each processor independently computing its load approximations G_i and checking that it is amongst the top p after adding x_j to the loads of machines not involved in the current BSP job; where $0 \leq x_j \leq 1$ is the contribution of the BSP process on machine j, to the load average on that machine. This calculation can be performed on entry to the superstep T seconds after the last checkpoint (i.e., this checking does not have to be performed in synchrony). The Boolean result from each of the processors is reduced with the or-operator to ensure that all processors agree to checkpoint during the same superstep. In the TCP/IP and UDP/IP implementations of *BSPlib*, this is piggy-backed onto a reduction that is used to globally optimise communication [3]. Therefore if a checkpoint is not necessary, there is no substantial impact on the performance of *BSPlib*.

[1] we distinguish between machines and processors as each machine may contain a number of processors, each of which runs multiple processes.

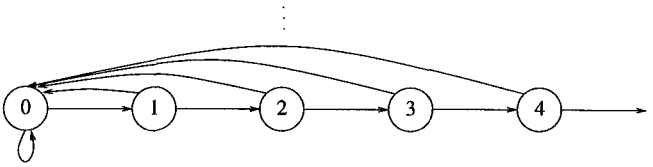

Fig. 1. Markov chain considering only single direct updates

4 Determining the Global Load of a System

The load daemons use a protocol in which the locally held global load states are periodically sent to k randomly chosen daemons running on other machines. This update period is uniformly and randomly distributed with each machine being independent. When a load daemon receives an update, it responds by merging the locally held load table and sending the resultant table back to the sender. The sender then merges the entries of the returned table with the entries of the locally held table. For purposes of simplifying the analysis, the updates are assumed to happen at fixed intervals and in a lockstep fashion. We also assume that the processors do not respond with the merged table, but merely update their tables by merging in the update requests. The analysis that follows always provides an upper bound for the actual protocol used.

If each processor sent out a message at the end of each interval to all the other processors, this would require P^2 messages to maintain the global state. A job requiring $p \leq P$ processes could contact P machines and arrive at an optimal choice of processors with considerably fewer messages provided that jobs did not start very often. However, with a large network of machines, the rate of jobs starting and the number of machines P would quickly lengthen the delay in scheduling a BSP job. By maintaining a global processor utilisation state at each of the machines, starting a job only involves contacting the local daemon when choosing a set of p processors and thus need not contribute to network traffic. Even once the ordering of processors has been selected, the problem of over assigning work to the least loaded machines can be avoided by having those machines reject the workload request based on knowledge built up locally. The algorithm then simply tries the machine with the next highest priority.

The quality of the decision for the optimal set of machines depends on the ages of the entries in the distributed load averages tables. If each machine uniformly and randomly chooses a partner machine at the end of each interval and sends its load average value to that machine, then the mean age of each entry in a distributed load average table can be calculated by considering the discrete time Markov chain shown in Figure 1. In this case there would only be p messages at the end of each interval, but the age distribution $\{\pi_i : i \in \mathbb{N}\}$, and the mean age μ are given by:

$$\pi_i = \frac{1}{p-1}\left(\frac{p-2}{p-1}\right)^i \tag{1}$$

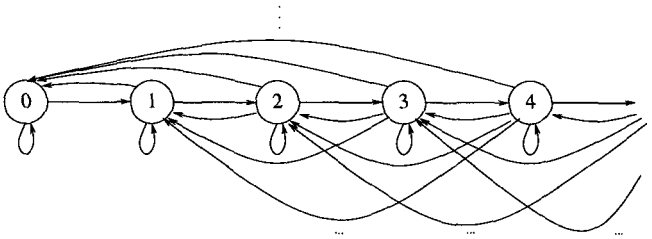

Fig. 2. Markov chain when indirect updates are allowed

$$\mu = \sum_{i=1}^{\infty} i\pi_i = p - 2 \tag{2}$$

By sending out messages more frequently, or sending out k messages at the end of each interval, the mean age can be reduced to $\mathcal{O}(p/k)$, but this increases the traffic and required number of virtual circuits to pk.

By allowing each machine to exchange *all* their load average information with k other machines at the end of each interval, a significant reduction in the mean age of the entries can be achieved with pk circuits. This scheme allows machines to choose between the most current load average figures, even if they were indirectly obtained from other machines. Figure 2 shows the corresponding Markov chain. In this stochastic process, transitions into state 0 can only arise out of direct updates, that is, a machine directly updating its entry in a remote table. The distribution, $\{\pi'_i : i \in \mathbb{N}\}$, of the ages of the entries in the load average table is given by the recurrence:

$$\pi'_i = \begin{cases} k/(p-1), & \text{if } i = 0 \\ \pi'_{i-1} P\{\textit{no useful updates}\} + \sum_{j=i}^{\infty} \pi'_j P\{\textit{min age is } i\}, & \text{otherwise} \end{cases} \tag{3}$$

Figure 3 shows the mean table entry ages of the three strategies when $k = 1, 3, 6$ against P. As P is given as a log scale, it is clear from the figure that while the first two strategies give a mean age μ as $\mathcal{O}(P)$, the third strategy (allowing indirect updates) gives a mean age of μ' as $\mathcal{O}(\log P)$.

If we replace the discrete times of the Markov chains with update intervals, t, the distributions above give the mean age at the beginning of each of the intervals. The figure shows that in order to bound the mean age to, say, five minutes we must ensure that:

$$t(\mu + \frac{1}{2}) \leq 5 \text{ minutes, or}$$

$$t \leq \frac{10}{2\mu + 1} \tag{4}$$

Therefore when $p = 32$, t should be less than or equal $3\frac{1}{3}$ minutes. The line marked "experimental results" shows the actual bounds on the algorithm for $t = 4$ minutes for all values of P. The experimental results are better than the upper bounds of the analysis as the updates don't occur in lock-step fashion, and a shorter sequence of updates are therefore possible. Also the actual protocol reuses the established circuit to send the merged tables back to the sender daemon;

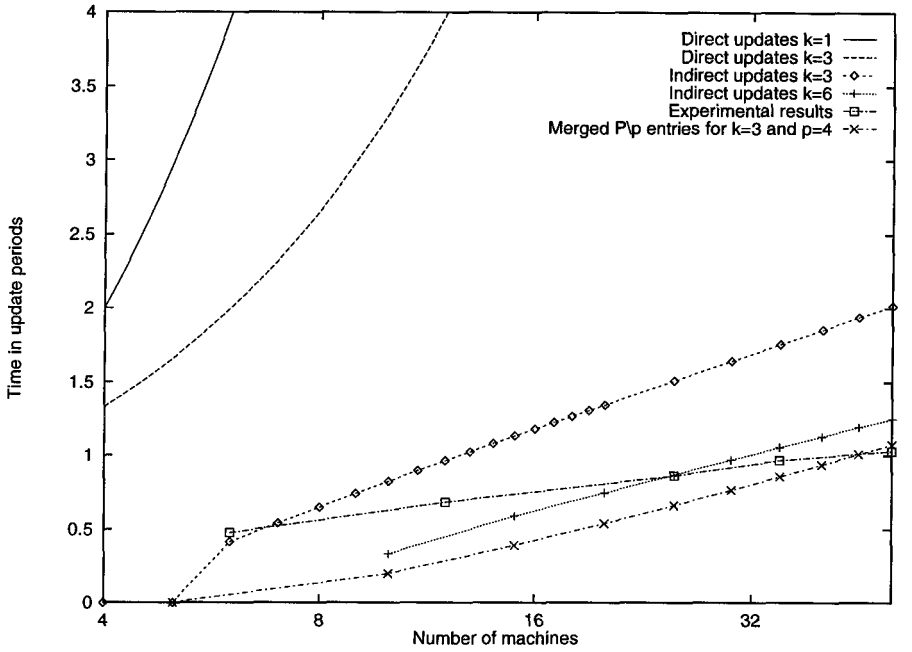

Fig. 3. Mean ages achieved by each of the three strategies and the merged $P \setminus p$ entries

this in effect allows the system to perform twice as many updates with fewer circuits.

As described in Section 3, by having the p processes involved in a BSP computation merge their load average tables before choosing where to migrate the processes, the *current* five minute load averages for the p processors executing the job is obtained and the ages of the load average table entries for the other $P - p$ machines have a distribution $\{\pi''_i : i \in \mathbb{N}\}$ where

$$\pi''_i = \sum_{x=1}^{p} \binom{p}{x} (\pi'_i)^x (1 - \sum_{t=0}^{i} \pi'_t)^{p-x} \tag{5}$$

Figure 3 includes the resulting mean age from this distribution for $p = 4$ against the total number of machines P, and compares it with the mean derived from the original distribution $\{\pi'_i : i \in \mathbb{N}\}$. It is clear from the figure that as p increases, the mean age of the data from the merged tables decreases until the mean age is zero when $p = P$.

5 Experimental Results for an Electro-Magnetics Application

The code EMMA T:FE3D (part of the British Aerospace EMMA electro-magnetic software suite), uses the finite element time domain method for solving Maxwell's

Table 1. Execution time in seconds for the electro-magnetics simulation

p	ypcat order	daemon order	daemon order + forced migration	daemon order + selective migration
2	3255	623	841	658
4	3855	1155	1916	1092
8	2968	1161		

equations in three dimensions. The finite element approach offers several advantages over other full wave solution methods (e.g., finite difference time domain, method of moments). A volume of space around the target is filled with an unstructured mesh of tetrahedra which can conform accurately to the geometry of the object being analysed and, because of the unstructured nature of the mesh, many small elements can be introduced in regions where the solution has rapid spatial variation. A time marching algorithm using a Taylor-Galerkin method is used to advance the fields through time to simulate the propagation of a wave through the mesh. The CFD community have developed considerable knowledge and expertise in unstructured mesh generation and finite element solvers which have been exploited for solving electro-magnetic problems. Applications areas for electro-magnetic solvers in the aerospace industry include electro-magnetic scattering, analysis of electro-magnetic compatibility and hazards, antenna design, and modelling of effects of lightning strike.

Table 4 shows the results from the electro-magnetics application running on various numbers of processors. The four columns of execution time show the following: (1) a job running on a random choice of machines. These may be highly loaded or may not have fast processors; (2) a job that is initiated on the p best machines (i.e., the fastest least loaded machines); (3) a job that is started on the best p machines, but is check-pointed every two minutes; and (4) a job that is started on the best p machines and checks every two minutes whether it is beneficial to check-point and migrate.

The first experiment is an approximation to a job that encounters poor service during execution. As can be seen from the dramatic decrease in execution times in the second experiment it is always beneficial to start a job on the most powerful unloaded machines. In the situation where the chosen processors have the same power, and the job is long running, then the second experiment will degrade to the performance of the first. The third experiment quantifies the overhead in check-pointing. As ten check-points were performed at $p = 4$ the increase in execution time shows that a local checkpoint takes 19 seconds to write a seven megabyte image to disc. The fourth experiment shows that there is little overhead in checking whether a check-point is necessary.

As it costs $19p$ seconds to perform a migration, it is only beneficial to migrate in situations where the processor usage is not too erratic, thus allowing the job to recoup the cost of the migration on the set of processors that were migrated to. If jobs are long running and compute bound then there is a lot of potential for regaining lost ground due to having to migrate from a loaded machine.

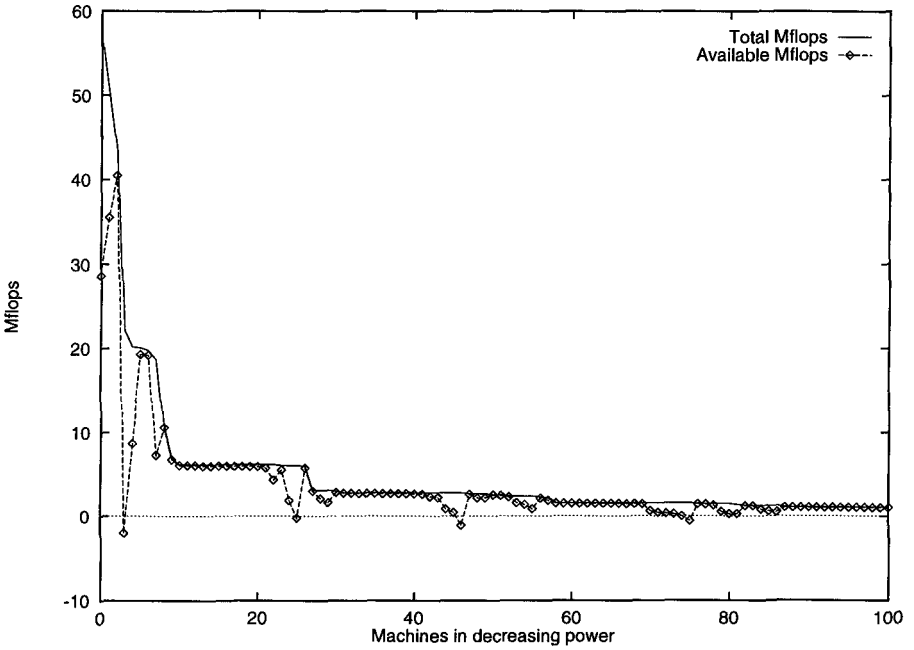

Fig. 4. Moore's law: a profile of the computational power of a collection of workstations

The results shown in Table 4 are from an electro-magnetics experiment that simulates a field around a sphere. As can be seen from the figure, no parallel speedup was achieved when increasing the size of the NOW; this was due to the dominance of communication over computation in this small test case. Larger realistic test cases at BAe have shown linear speedup up-to sixteen processes as computation begins to dominate.

This work is based on the assumption that the available cycles on the machines changes continuously over time. However, with a large resource acquired over time, the machines are unlikely to be homogeneous in available power. Figure 4 shows a graph of available computing power on each workstation in Oxford University Computing Laboratory against available power at a particular instance in time. We express available power by the formula $(n - L)s$; where n is the number of processors in a machine, L is the load average on that machine and s is the Mflops/s rating of a single processor. The graph demonstrates Moore's Law in the purchase of workstations over time, i.e., to the right of the graph, there are a large number of low powered aging workstations, whereas toward the left of the graph, there are few high performance machines procured over the last six months. When choosing to schedule a p process parallel job, either a homogeneous set of unloaded (slow) machines can be used, or the best p. The policy we adopt is that although the unloaded machines ensure little interference, they only provide a fraction of the power of the best machines. However, running on the popular powerful machines, can also make a job susceptible to low throughput as the available cycles at a node can vary dramatically over time.

For example, the figure shows that the fourth machine from the left has a peak performance of 50 Mflops/s, yet it is so loaded that there are no free cycles for parallel jobs. In summary, the erratic, but powerful machines, to the left of the graph are most suited to computational intensive parallel jobs, yet they are the very machines that require process migration.

6 Related Work

The process migration work described in this paper, also provides fault tolerance if the mean time between failure is greater than the rate at which processes migrate/checkpoint. Existing work in this area has tended to concentrate on fault tolerance using redundant computation. For example Nibhanupudi and Szymanski [9] minimise the slowdown of a BSP job when external loads are applied to machines in a network by replicating computation on a number of machines. At the end of a superstep, the results of the fastest of the replicated jobs is used to form the global state of the system. Although their system allows the mean time between failure to be less than ours due to the replicated jobs, their prototype system requires user annotation of the data structures to be included in a checkpoint, and they assume that there won't be a machine failure during communication. In contrast, the fault tolerance in our system is transparent to the user, and has no restrictions on when faults can occur. Also, our approach doesn't suffer from the considerable overhead that would be incurred to implement a process replication scheme. As already noted, the only overhead we incur is the time taken to write the p checkpoints to disk. A similar approach to ours is that of Kaashoek *et al.* [8] where fault tolerance of Orca programs is provided on top of the Amoeba distributed operating system. Their approach is complicated by the fact that they have to determine locally when communication is quiescent so that a stable checkpoint can be taken. Other parallel check-pointing systems for MPI [12] and PVM [13] also suffer from this problem, as a checkpoint can only be performed if there is no communication in transit when each process performs a local checkpoint to capture the global state [2]. Fortunately, the check-pointing regime described here is far simpler than any of the approaches used in message passing systems as opportunities for a global checkpoint naturally arise out of the superstep structure of BSP programs. The process migration facilities provided for MPI [12] and PVM have usually been developed on top of the check-pointing and batch scheduling facilities provided by Condor [4] and LSF[15].

7 Conclusions

We have shown that it is possible to perform fault tolerance and process migration of BSP programs in a transparent way on a network of workstations. By paying careful attention to the design of a distributed load manager, it is possible to determine the global load of a system with minimal impact on network

traffic. This, in conjunction with the pro-active manner in which BSP jobs migrate between machines, enables a system that is unobtrusive to non-BSP users, whilst providing the best of the resource as a whole.

References

1. David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. Nas parallel benchmarks 2.0. Technical Report 95-020, NAS Applied Research Branch (RNR), December 1995.
2. R. Baldoni, J. M. Hélary, A. Mostefaoui, and M. Raynal. A communication-induced checkpointing protocol that ensures the rollback-dependency trackability property. In *Proc. of the 27th IEEE Symposium on Fault-Tolerant Computing Systems (FTCS)*, pages 68–77, Seattle, WA, June 1997. IEEE.
3. Stephen R. Donaldson, Jonathan M. D. Hill, and David B. Skillicorn. Predictable communication on unpredictable networks: Implementing BSP over TCP/IP. In *EuroPar'98*, LNCS, Southampton, UK, September 1998. Springer-Verlag.
4. D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A worldwide flock of condors : Load sharing among workstation clusters. *Future Generations of Computer Systems*, 12, 1996.
5. Jonathan M. D. Hill, Stephen Jarvis, Constantinos Siniolakis, and Vasil P. Vasilev. Portable and architecture independent parallel performance tuning using a call-graph profiling tool. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*, pages 286–292. IEEE Computer Society Press, January 1998.
6. Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, to appear 1998. see `www.bsp-worldwide.org` for more details.
7. Jonathan M.D. Hill, Stephen Jarvis, Constantinos Siniolakis, and Vasil P. Vasilev. Analysing an sql application with a bsplib call-graph profiling tool. In *EuroPar'98*, LNCS, Southampton, UK, September 1998. Springer-Verlag.
8. M. F. Kaashoek, R. Michiels, H. E. Bal, and A. S Tanenbaum. Transparent fault-tolerance in parallel orca programs. In *Proc. Symp. on Experiences with Distributed and Multiprocessor Systems III*, pages 297–312, 1992.
9. Mohan V. Nibhanupudi and Boleslaw K. Szymanski. Adaptive parallelism in the bulk synchronous parallel model. In *EuroPar'96*, number 1124 in Lecture Notes in Computer Science, pages 311–318, Lyon, France, aug 1996. Springer-Verlag.
10. David Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, Fall 1997.
11. Richard M. Stallman. Emacs: The extensible, customizable, self-documenting display editor. AI memo 519A, Artificial Intelligence Laboratory, Massachusetts Institute of Technology (MIT), 1979.
12. G. Stellner. CoCheck: checkpointing and process migration for MPI. In IEEE, editor, *Proceedings of IPPS '96. The 10th International Parallel Processing Symposium: Honolulu, HI, USA, 15–19 April 1996*, pages 526–531, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.
13. Kasidit Chanchio Xian-He Sun. Efficient process migration for parallel processing on non-dedicated networks of workstations. Technical Report TR-96-74, Institute for Computer Applications in Science and Engineering, December 1996.

14. Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
15. Jingwen Wang, Songnian Zhou, Khalid Ahmed, and Weihong Long. LSBATCH: A distributed load sharing batch system. Technical Report CSRI-286, Computer Systems Research Institute, University of Toronto, April 1993.