Proceedings of the 1996 Winter Simulation Conference
ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain

DYNAMIC MEMORY USAGE IN PARALLEL SIMULATION: A CASE STUDY OF A LARGE-SCALE MILITARY LOGISTICS APPLICATION

C. J. M. Booth
D. I. Bruce
P. R. Hoare
M. J. Kirton
K.R. Milner
I. J. Relf

Defence Research Agency,
St. Andrew's Road, Malvern,
Worcestershire WR14 3PS, UNITED KINGDOM.

ABSTRACT

We report on a comparative study of the performance of shared and distributed memory parallel simulation algorithms on a large-scale military logistics simulation, and describe the nature of the application and its parallelisation in some detail. We demonstrate that the patterns of communication in the simulation were such that a standard implementation of Breathing Time Buckets (BTB) was unable to achieve any speed-up. New variants of BTB were designed and implemented, and we were able to achieve a speed-up of 7.4 compared to a critical path limit of 16.9. The logistics simulation contained a number of complex data structures and its parallel implementation was highly memory-intensive. The resulting patterns of memory access significantly degraded the shared memory simulation performance relative to the equivalent distributed memory version. These results cast doubt on the effectiveness of the current generation of shared memory parallel computers in dealing with optimistic simulations of large, dynamic scenarios.

1 INTRODUCTION

The emergence of relatively low-cost shared memory parallel computers has been a significant development in the last few years. Many manufacturers now offer multi-processor workstations with a global memory model. The advantage of a single address space is that data is immediately visible to each processor and message-passing can be implemented by simply passing data references. Although shared memory machines have distinct advantages over distributed memory ones, developing efficient shared memory simulators presents a number of problems, particularly for memory-intensive applications.

The logistics application consisted of about 16,000 lines of C++ code and simulated the activity of the British Army's Royal Logistics Corps during the first few days of a military campaign. There were three main features associated with the logistics application that made it worthy of detailed study in the context of parallel simulation. First, it was highly memory-intensive: the state associated with many of the simulation objects contained dynamic data structures of considerable complexity, thus generating a large overhead associated with saving and restoring state. Second, it exhibited poor lookahead by repeatedly generating events in the near future. Finally, there were a number of 'self-propelled' objects—objects that moved rapidly ahead in simulation time independently of all other objects in the system, needing to be repeatedly rolled back unless the optimism in the underlying simulator were constrained in some way.

A major aim of our research programme was to build general-purpose simulators capable of dealing with fully-interconnected objects with no pre-determinable lookahead. For these reasons, our work was based on optimistic simulation protocols: BTB (Steinman 1992) and Time Warp (Jefferson 1985). Our simulators were coded in C++ and all implemented the same simulation programmer's interface (SPI). Our shared and distributed memory computers were both SPARC-based and ran the same operating system (Solaris 2.3), enabling us to make a direct comparison between simulators.

The remainder of this paper is arranged as follows. In section 2, we describe the logistics simulation. Our shared and distributed memory simulators are described in sections 3 and 4, respectively; extensions to the standard BTB algorithm are detailed in section 5. We present our results in section 6 and summarise our work in section 7.

2 LOGISTICS APPLICATION

2.1 Major functional components

Our logistics model (Hoare et al. 1995) simulated the deployment of equipment during a military campaign. The

re-supply model consisted of fixed first-line storage areas (HQ and Division), intermediate storage areas (Brigade and Battle Group) and front-line (Unit-level) dumps connected by a network of links as shown in Figure 1.

Storage areas consisted of a number of separate areas called circuits in which stacks of stores were arranged in rows. Trucks in a given storage area traversed the stacks collecting stores and were then assembled into convoys for dispatch to a target storage area or dump where they would unload. A dump consisted of a number of trucks in the process of being unloaded as stores were required. Trucks returned to their starting point when they were empty. The base personnel were also modelled in that they were required to rest for a minimum number of hours per day.

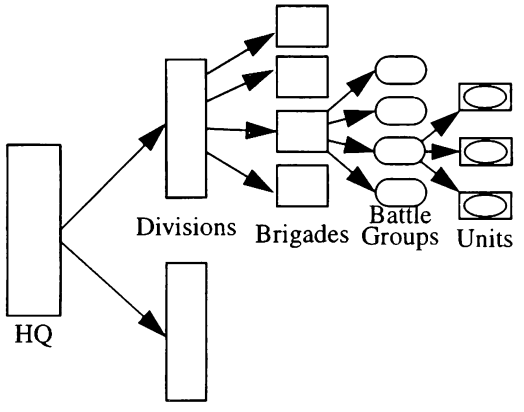

Figure 1: The Basic Layout of the Logistics Scenario

The model operated on a daily cycle; at the beginning of the simulation half of the trucks in the storage areas were already loaded awaiting dispatch. For each simulated day, the loaded trucks were dispatched to destinations to satisfy actual requirements and the remaining empty trucks were loaded with the predicted requirements for the next day. In the results we report on, the model was simulated for a period of two days.

We used two configurations based on Figure 1 which we will refer to as single-division and twin-division, in which the single HQ provided orders for one and two hierarchies respectively.

2.2 Simulation Objects

The storage areas and dumps were treated as simulation objects, and each had a significant amount of associated state and processing. For the results reported in section 6, the simulation objects were distributed across the processors in a round-robin fashion.

A major factor affecting the writing and debugging of large-scale optimistic simulations is state saving. The logistics application contained a number of complex data structures which needed to be recoverable in the event of a rollback. Neither full state saving nor incremental state saving offered attractive solutions—in particular, it was not possible to determine before an event was processed just how many elements in a particular data structure would change.

In order to minimise the programming effort associated with the saving of state, we developed a prototype data structure manager which maintained its own time-dependent state information. Its implementation as a C++ class hid the details from the model writer but was only a half-way solution to the problem as it only dealt with list types. We shall report elsewhere on our results using a version of logistics in which state saving was carried out transparently for all data types using persistent data structures (Bruce 1995).

2.3 Event Distributions

Within the hierarchical structure depicted in Figure 1, a simulation object only communicated externally with its immediate ancestor and descendants; communication between objects at the same level in the hierarchy did not take place. Excluding start-up messages, the ratio of intra- to inter-object messages was about 5:1.

Many of the events were concerned with the movement, loading and unloading of trucks around the Battle Group storage areas. In fact, for the single-division configuration, these types of event that the sixteen Battle Groups scheduled for themselves accounted for about 45% of the total number of generated messages.

Each of the 48 unit-level storage dumps independently generated a re-order event for itself every simulated half-hour. The unit-level storage dumps were therefore essentially self-propelled objects.

The major interaction between storage areas at different levels arose from the sending and returning of trucks: on the outward journey the trucks were filled with supplies, unloaded at pre-determined stacks and returned to the storage area that dispatched them. At the unit-level dumps, supplies were consumed at pre-determined rates. The units could also send urgent requests to the next level up in the hierarchy for supplies.

In Figure 2, we present the measured distribution of lookahead times for the single-division scenario, where we define the lookahead to be the difference between the receive and send times of an event. Over half of the events were generated with time stamps less than two simulated minutes into the future. At an even finer level of granularity, nearly ten thousand events were scheduled with times less than 0.01 simulated minutes. The events at thirty minutes corresponded to the re-order events fired off by the unit-level storage areas.

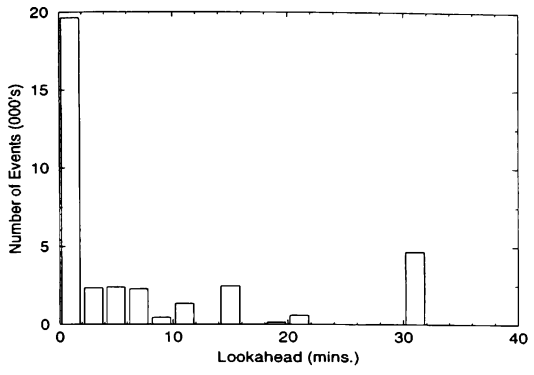

Figure 2: Distribution of Lookahead Times for the Single-Division Configuration

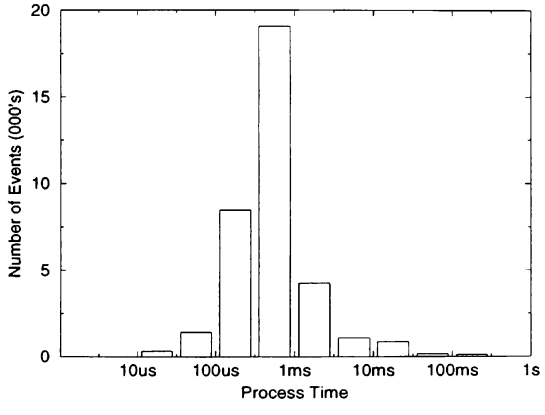

Figure 3: Distribution of Event Processing Times for the Single-Division Configuration

Figure 3 shows that the distribution of event processing times was very broad—ranging from about 10 μs to 1.0s—though the overwhelming majority resided in the range 0.1—10 ms. The larger granularity events were involved in searching a storage area for supplies—this search involved iterating over heavily-nested loops. The granularity of these types of event was very data-dependent and also led to the consequence that an event processed for the second time following a rollback sometimes required a significantly different amount of CPU time than was needed to process it initially.

Finally, we note that many of the events in the logistics simulation needed to pass lists of data values (for example, a truck contained a list of the pallets it was carrying). Due to the inherent difficulties in passing dynamic data structures between different memory spaces, these lists were implemented as fixed-size arrays. This made the copying and movement of the events straightforward, but added to the memory consumption of the simulation. In practice, we found the event sizes to be grouped around 1 Kbyte, 8 Kbytes and 16 Kbytes.

3 SHARED MEMORY SIMULATORS

3.1 Hardware

The shared memory computer we used was an 8-processor SPARCserver 1000 (SS1000) manufactured by Sun Microsystems. It contained 50 MHz SuperSPARC processors with 1 Mbyte of external cache per processor and a 64-bit wide packet switched XDBUS operating at 40 MHz, giving a peak bandwidth of 320 Mbytes/s to its 256 Mbyte memory. The Solaris threads library provided the necessary parallel programming support.

3.2 Memory Management

Our early performance experiments revealed that the dynamic memory routines of the compiler scaled very poorly. Since our parallel simulators made considerable use of dynamic memory we found it necessary to implement our own dynamic memory algorithm, which we did by means of a C++ `MemoryManager` class. Moreover, these experiments demonstrated the significant effect data locality could have on performance cf. (Fujimoto and Panesar 1995), and thus each processor had its own `MemoryManager`.

In the interests of access speed, memory within a `MemoryManager` was maintained using a set of free-lists. We used a BSD-style algorithm, in which a free-list was maintained for each power of 2. Memory returned by the `MemoryManager` was also aligned on cache-line boundaries and no block returned was smaller than the cache-line size (64 bytes in our case). These restrictions were designed to prevent the problems associated with false sharing (Torrellas et al. 1994) that can occur with multi-processor cache-based machines. Dynamic memory management on a given processor was also completely independent of all other processors: there were no shared variables and no concatenation of blocks or merging of free-lists. In addition, free-lists operated a 'last-in first-out' policy, which meant that memory returned from a free-list had a good chance of residing in the local cache of its processor.

In our current version, the `MemoryManager` class had no mechanism for preventing a net transfer of memory from one processor to another, which could be a problem if the simulation contained net source or sink objects.

3.3 Breathing Time Buckets

Despite requiring synchronisation at the end of each cycle, the Breathing Time Buckets (BTB) algorithm (Steinman 1992) has the advantages of simplicity coupled with a certain elegance and robustness. BTB consists of a number of distinct phases: event processing and local event horizon (LEH) exchange; global simulation time (GST) determination; event committal and exchange; sorting and merging event queues; local rollback and housekeeping. On the SS1000, we used the Solaris threads library to bind one 'main loop' thread to each processor.

Each processor maintained its own separate event queue and was responsible for a subset of simulation objects. Output messages from a given processor were inserted directly into the input queues of the appropriate nodes. GST was determined via a set of shared variables arranged in a tree structure. As each processor crossed its LEH it informed its parent of the crossing and wrote its LEH value into a single global variable if it was less than the value recorded thus far. Once the root node was informed that all LEH's were crossed, it set the global Boolean `allCrossed` to true, allowing all processors to move on to the next phase of the BTB cycle. We used a tree-based algorithm to ascertain when event exchange was completed, since this was substantially faster than using a shared counter.

3.4 Time Warp

Our implementation of Time Warp was similar in spirit to the BTB implementation described above. A 'main loop' thread was run by each processor. Again, there was no global input queue; output messages were inserted into an unsorted input buffer residing on the receiving processor. The input buffer was scanned by the receiver and messages extracted and placed into its pending queue in time stamp order. After an event was processed, it was inserted into the processed event queue associated with its simulation object. Rollback was implemented using direct cancellation (Fujimoto 1989).

We used a version of the global virtual time (GVT) algorithm described in (Fujimoto and Hybinette 1994). This algorithm makes use of the fact that on a shared memory computer there are no messages in transit and has the advantage of being able to operate without closing the simulation down. Also, there is no need to acknowledge messages or perform rounds of token passing.

4 DISTRIBUTED MEMORY SIMULATORS

4.1 Hardware

We used a 22-processor Meiko CS-2 containing 50 MHz SuperSPARCs with 1 Mbyte of external cache and 64 Mbytes of memory per processor. Two of the processors were I/O nodes with file systems mounted and connections to ethernet; the remaining 20 processors effectively operated as diskless workstations served by the I/O nodes. Each processor had its own dedicated interface to a high-performance communications network—a Meiko designed communications processor named Elan—which provided 50 Mbytes/s bisectional bandwidth over a physical link operating at 0.6 Gbit/s in each direction. For a 64 byte message, the latency (including transfer) and effective bandwidth between UNIX processes on different processors was 12 μs and 5.54 Mbytes/s; for 64 Kbyte messages the corresponding figures were 1.491 μs and 43.95 Mbytes/s.

The CS-2 also supported a global data space, and very efficient message transfers were achievable by the direct memory access (DMA) transfer procedures supported by the Elan processor. Also, there was direct support in hardware for processor synchronisation.

4.2 Breathing Time Buckets

The parallel BTB program comprised the same executable running on every processor. The BTB processes interacted via message passing and hardware synchronisation. In order to perform the LEH broadcast as efficiently as possible, we used the global memory provided by the CS-2. Thus when a processor crossed its LEH, it broadcast this value via low-level DMA's to all other processors. When all LEH's were crossed, each BTB process was able to determine independently the global simulation time.

During the event committal stage, each BTB process stored its generated events and placed them in an array of message buffers, according to their destination. Message sending was accomplished using the Elan Channel Protocol, which provided a fully robust yet very efficient technique for sending variable sized messages. After sending its messages, each BTB process performed a hardware synchronisation and blocked until all processors reached this point.

5 EXTENSIONS TO BTB

During the course of our work on logistics, we developed three variants on the basic BTB algorithm, which we named *limited*, *extended* and *limited-extended*. *Limited* BTB stops processing events as soon as its local event horizon (LEH) is crossed. This can significantly reduce the

number of potentially costly rollbacks but reduces the optimism of the simulator. *Extended* BTB folds back into the event queue events generated for objects on the same processor. Such events are not in general considered for the LEH calculation and for applications like logistics, this can greatly reduce the number of BTB cycles required for a given end simulation time.

There is a slight subtlety in the implementation concerning the way in which *extended* BTB interacts with optimistically processed events. A processor may continue to process beyond its LEH in the hope that optimistically processed events will not need to be rolled back. Unfortunately, a local event may cause one of these optimistically processed events to roll back; if this is the case, the local event *must* contribute to LEH—if it did not, then any optimistically processed events that should have been rolled back may be incorrectly committed.

Another consequence of *extended* BTB is that in simulations which consist entirely of local events, no BTB cycles will be triggered and therefore no dynamic memory will be recovered. *Extended* BTB therefore requires an additional synchronisation mechanism, akin to GVT in Time Warp, in which memory recovery is forced at regular intervals.

As its name suggests, *limited-extended* folded events back into the event queue but stopped processing after the LEH was crossed.

6 RESULTS

The following results were obtained using simulators that had already demonstrated good speed-up on the queuing network application described in (Damitio et al. 1994). For example, the distributed memory BTB simulator achieved a maximum speed-up of 17.5 on 20 processors and achieved significant speed-ups over a range of parameters. There was also no significant difference in performance between the shared memory and distributed memory simulators using the synthetic application.

All of the speed-up figures quoted in the results below were calculated relative to an optimised sequential simulator.

6.1 Distributed Memory

The speed-up figures obtained for the two logistics scenarios (single-division and twin-division) on the CS-2 are shown in Figures 4 and 5 respectively. Standard BTB was unable to achieve any speed-up on the single-division configuration (and this result was repeated on the SS1000). The main reason for this was the large number of rollbacks that occurred as a result of objects sending messages a short distance into the future. These messages often triggered costly rollbacks, severely degrading the overall performance.

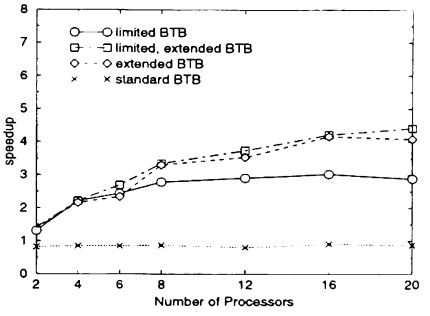

Figure 4: Speed-up for the Single-Division Logistics on the CS-2

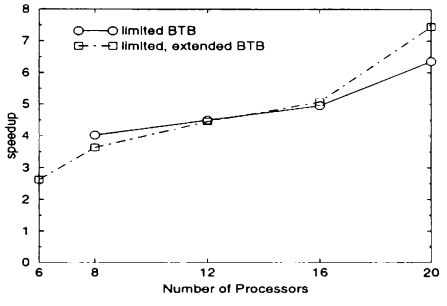

Figure 5: Speed-up for the Twin-Division Logistics on the CS-2

Overall, the best speed-up was obtained with *limited-extended* BTB; a speed-up of 4.1 was achieved on the single-division logistics (out of a maximum possible speed-up of 8.1) and 7.4 achieved on the twin-division scenario (out of a maximum of 16.9). Our maximum speed-up figures were obtained using a critical-path analysis (Wieland et al. 1992) in which the best possible completion time for a simulation was calculated (neglecting state-saving and message-passing overheads). The twin-division scenario was so memory-intensive that the *extended* BTB simulator would not run it to completion and the *limited* and *limited-extended* simulators could only do so using at least 8 and 6 processors respectively.

6.2 Shared Memory

The speed-up curves for single-division and twin-division logistics simulations on the SS1000 are shown in Figures 6 and 7 respectively. If we compare these results with those obtained on the CS-2, we see that the speed-ups achieved on 8 nodes were significantly worse than the equivalent CS-2 values. For single-division logistics, the speed-up was 2.3 on the SS1000 compared with 3.3 on the CS-2; for the twin-division we obtained a speed-up of 2.7

on the SS1000 and 4.0 on the CS-2. In terms of run-time, the distributed memory simulator consistently outperformed its shared memory counterpart by about 50%.

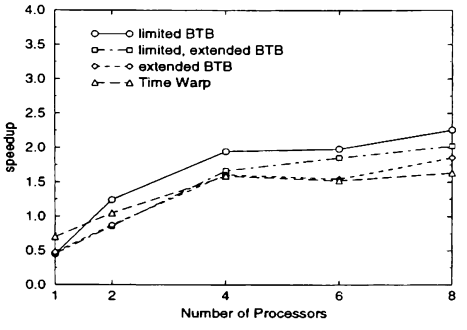

Figure 6: Speed-up for the Single-Division Logistics on the SS1000

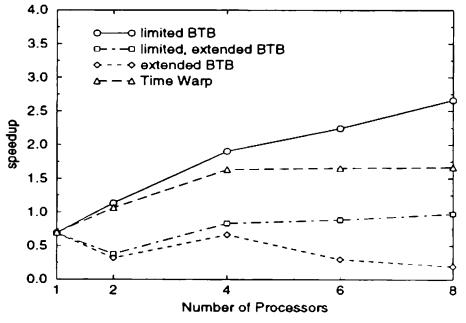

Figure 7: Speed-up for the Twin-Division Logistics on the SS1000

A breakdown of overheads for *limited* BTB is shown in Figure 8. 'wait LEH' represents the waiting time at the end of a cycle when a processor has reached its LEH, and is waiting for all other processors to synchronize; 'save state' is the time spent saving the state associated with each event; 'commit' includes memory recovery; the main constituents of 'housekeeping' are the merging and sorting of pending event queues; and 'wait exchange' is the time spent awaiting messages from other processors at the end of a cycle.

Much of the overhead associated with BTB was in the wait LEH time at the end of a cycle. The different BTB variants also exhibited quite different behaviour for the logistics application. *Limited* BTB (Figure 8) generated just over 1800 BTB cycles in the example shown and had no rollbacks; a substantial amount of time was spent waiting at the end of a cycle, but no time was wasted unnecessarily processing events. *Extended* BTB generated just 40 cycles for the same simulation, and it spent much less time waiting for inter-processor synchronisation; however, it also spent a considerable amount of time processing events that were subsequently rolled back (this was roughly 20% of the total number of events for *extended* BTB). The overall effect was that *limited* BTB consistently outperformed *extended* BTB.

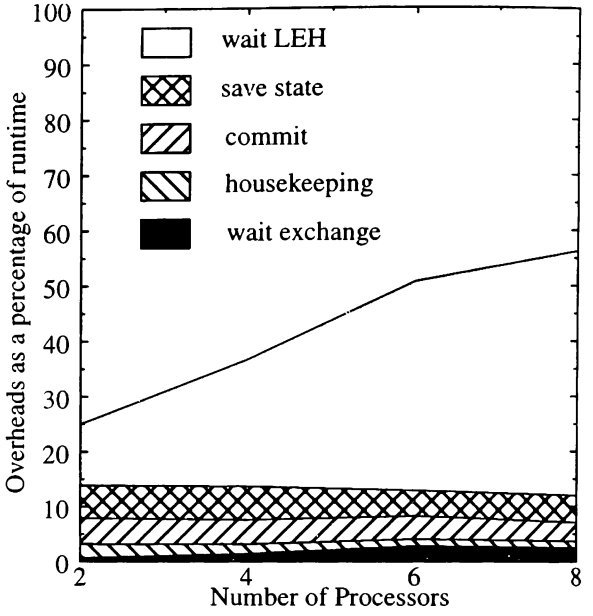

Figure 8: Breakdown of Overheads for the Single-Division Logistics Simulation on the SS1000 using *limited* BTB

6.3 Degradation of event processing time

When we analysed the breakdown of the overall run-time, we discovered that on the SS1000, the average time to process an event degraded significantly with increasing numbers of processors (Figure 9). Similar degradation was not observed on the CS-2 (Figure 10) and this is the main reason for the performance difference between our shared and distributed memory simulators.

There is also a difference between the event processing time for the sequential simulator and the parallel simulators running on one node. This is due to the overhead of the state-saving implementation, which is absent for the sequential simulator. Also, memory access times were slightly better on the CS-2, compared to the SS1000, which meant that the absolute run-times for the sequential simulator were consistently 10% less than on the SS1000.

Before we consider possible explanations for the degradation in event processing time on the SS1000, we should note that this behaviour is observed both for BTB and Time Warp; it is not specific just to BTB (or its variants). The event processing time is being averaged over committed events, so that a comparison is being made over exactly the same set of events. Also, the memory management allocator used was designed to avoid false sharing effects, in which two or more independent varia-

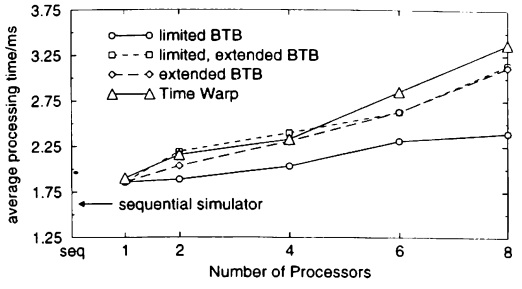

Figure 9: Average Event Processing Time for the Single-Division Logistics Simulation on the SS1000

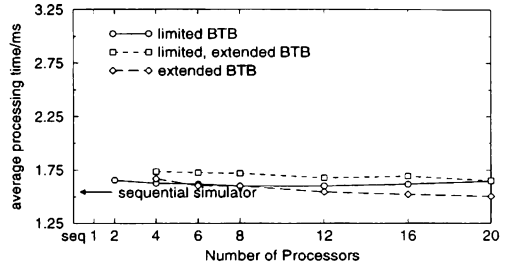

Figure 10: Average Event Processing Time for the Single-Division Logistics Simulation on the CS-2

bles share the same cache line. Moreover, it is worth bearing in mind that our synthetic queuing network application did not show any such degradation in event processing time.

Our hypothesis is that the shared memory performance is being affected by the bus traffic generated, since logistics is a very memory-intensive application, typically consuming 50 Mbytes or more in a typical run. One of the difficulties in programming shared memory machines is in analysing and diagnosing performance bottlenecks. Unfortunately, reliable and accurate tools for monitoring cache miss rates were not available, and therefore in order to test whether it was the bus traffic causing the problem we introduced random delays into our shared memory BTB simulator, immediately prior to an event being processed. The rationale here is that as the delay was progressively increased, the bus traffic density would be correspondingly reduced, as each processor spent more and more time waiting. The results of applying delays to the SS1000 BTB simulator are shown in Figure 11.

The average processing times for *extended* and *limited*-extended decreased significantly as the wait time was increased. This effect was less marked for *limited* BTB, but under normal conditions (i.e. without any artificially introduced waiting time), *limited* spends considerably longer waiting for synchronisation at the end of a cycle, compared to either *extended* or *limited-extended*. Under normal conditions it would therefore generate less bus traffic and hence be less prone to degradation in event processing time than either *extended* or *limited-extended*.

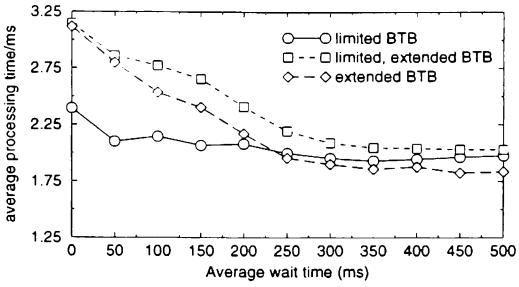

Figure 11: Average Processing Time as a Function of Wait Time on the SS1000 (all data points were generated using 8 processors with wait times normally distributed and variance = mean/2)

The explanation for the lack of degradation in event processing time for our synthetic queuing application (Damitio et al. 1994) is that the spin loop at the core of the event processing procedure did not generate any bus traffic. In fact, by artificially increasing the state size of each server object and modifying its event processing, we were able to demonstrate the same effect observed with the logistics application.

7 CONCLUSIONS

We have presented a study of the application of parallel simulation technology to a large-scale military logistics simulation and compared its performance using both shared memory and distributed memory computers. We have developed extensions to the basic BTB algorithm (*limited*, *extended* and *limited-extended*) and have managed to exploit roughly half of the available parallelism for a large-scale logistics simulation on the CS-2.

Comparing our shared and distributed memory simulators, we have shown that with a memory-intensive application like logistics, the speed-up obtainable can be affected significantly by specifically shared memory effects. Since optimistic simulations tend to make heavy use of dynamic memory, these results would seem to cast doubt on the capability of the current generation of multiprocessor workstations in dealing with optimistic simulations of large, dynamic applications.

ACKNOWLEDGEMENTS

It is a pleasure to thank Brian Roberts and Brian Merrifield for their long-standing support and encouragement.

REFERENCES

Bruce, D. 1995. "The treatment of state in optimistic systems", pp 40–49 in *Proc. Ninth Workshop on Parallel and Distributed Simulation*, Lake Placid, New York, 14–16 June 1995.

Damitio, M. et al. 1994. "Comparing the breathing time buckets algorithm and the Time Warp Operating System on a transputer architecture", pp 141–145 in *Proc. Modelling and Simulation (SCS European Simulation Multiconference)* 1994, Barcelona, Spain, 1–3 June 1994.

Fujimoto, R. M. 1989. "Time Warp on a shared memory multiprocessor", *Transactions of the SCS*, Vol. 6, No. 3, July 1989.

Fujimoto, R. M. and M. Hybinette. 1994. "Computing global virtual time in shared memory multiprocessors", Georgia Institute of Technology Technical Report, August 1994.

Fujimoto, R. M. and K. S. Panesar. 1995. "Buffer management in shared memory Time Warp systems", pp 149–156 in *Proc. Ninth Workshop on Parallel and Distributed Simulation*, Lake Placid, New York, 14–16 June 1995.

Hoare, P. R. et al. 1995. "The application of high performance parallel computing to military simulation", pp 115–119 in *Proc. Military, Government and Aerospace Simulation Conference*, Phoenix, Arizona, 9–13 April 1995.

Jefferson, D. R. 1985. "Virtual time", pp 404–425 in *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, July 1985.

Steinman, J. S. 1992. "SPEEDES: A multiple-synchronization environment for parallel discrete-event simulation", pp 251–286 in *International Journal of Computer Simulation*, Vol. 2, No. 3, 1992.

Torrellas, J., M. S. Lam and J. L. Hennessy. 1994. "False sharing and spatial locality in multiprocessor caches", pp 651–663 in *IEEE Transactions on Computers*, Vol. 43. No. 6, June 1994.

Wieland, F. et al. 1992. "A critical path tool for parallel simulation performance optimization", pp 196–206 in *Proc. 25th Hawaii International Conference on System Sciences*, Vol. II, Kauai, Hawaii, 7–10 January 1992.

AUTHOR BIOGRAPHIES

CHRIS BOOTH has a BA in Mathematics (1986) and an MSc in Computation (1987) from the University of Oxford. He is interested in adaptive synchronisation algorithms for PDES, and is currently working on the APOSTLE simulation language.

DAVID BRUCE has a BSc in Mathematics (1988) from the University of Kent at Canterbury. His research interests include parallel discrete event simulation, state saving for optimistic computation, programming languages and type systems and he is currently working on the APOSTLE simulation language and the DoD High Level Architecture (HLA) for simulation.

PETER HOARE has a Computer Science BSc and PhD from Royal Holloway, University of London. His research interests currently include distributed/parallel simulation, computer graphics and Internet protocols and systems. He is involved in the UK STOW programme as the technical lead on the DoD HLA.

MICHAEL KIRTON has BSc and PhD degrees in Physics, and before joining DRA held an IBM (UK) Fellowship. He has published over thirty scientific papers in fields ranging from noise in solid-state microstructures to parallel simulation.

ROY MILNER has a BSc in Mathematics (1983) from the University of Nottingham. His research interests include PDES on shared memory machines and he is currently working on the DoD HLA.

IAN RELF has a BEng in Engineering Electronics from the University of Warwick (1989). His research interests include the application of PDES to military simulations. He is currently working on a large-scale DIS air-ground simulation.