An Empirical Comparison of Runtime Systems for Conservative Parallel Simulation

Chu-Cheow LIM* Yoke-Hean LOW* Wentong CAI+
Wen Jing HSU+ Shell Ying HUANG+ Stephen J. TURNER×

Abstract

A main consideration when implementing a parallel simulation application is the choice of the parallel simulation protocol (conservative vs. optimistic). Given a particular protocol, the application programmer then has to determine a suitable parallel runtime system to implement the application. If the choice is an optimistic protocol, there are several parallel simulation libraries intended for application programmers (e.g. GTW, Warped). For a conservative protocol, the most effective approach is for the programmer to use a general parallel runtime library, and implement optimizations specific to the simulation application and/or model. In this paper, we selected four general parallel runtime libraries potentially relevant to parallel simulations, and implemented a conservative protocol on each of them. We study the four libraries on three main aspects: (a) programmability; (b) performance, and (c) mechanisms for performance tuning. Our target platforms are machines supporting shared address spaces (e.g. SGI Origin200, Sun Enterprise 3000), and we obtained performance figures from a 4-CPU Ultra2 Sun Enterprise 3000. From our experiments, we find that POSIX, though an industry standard, still has relatively high overheads, and cannot efficiently support a protocol with fine-grain LPs. The research libraries all show speedups on 4 processors, but to different extents. Cilk speedup curves improves with larger thread granularity, while Active threads show relatively good speedup even for small thread granularity. BSP processes are naturally coarse-grained, and thus good speedup is achieved in our simulation application.

1. Introduction

The background of this work is from an ongoing collaborative project between the Gintic Institute of Manufacturing Technology and the School of Applied Science in Nanyang Technological University, Singapore. The objective of our project is to study how parallel and distributed simulation techniques can be applied in a virtual factory simulation [9]. The simulations will be plant-wide, and include the modeling of manufacturing and business processes, and communications network. Such a simulation environment will allow one to model and analyze the effects of different system configurations and control policies on actual system performance. The initial focus is the electronic industry, because it is a major contributor to the manufacturing sector in Singapore.

A main consideration when implementing a parallel simulation application is the choice of the parallel simulation protocol (e.g. variants of conservative vs. optimistic protocol [5]). Given a particular protocol, the application programmer then has to determine a suitable parallel runtime system to implement the application. For a conservative protocol, the most effective approach is for the

---

* Gintic Institute of Manufacturing, 71 Nanyang Drive, Singapore 639798.
Emails: {cclim,yhlow}@gintic.gov.sg
+ School of Applied Science, Nanyang Technological University, Singapore 639798.
Emails: {aswcai,ashsu,assyhuang}@ntu.edu.sg
× Dept. of Computer Science, University of Exeter, EX4 4PT, Exeter, U.K. Email: steve@dcs.exeter.ac.uk

programmer to use a general parallel runtime library, and implement optimizations specific to the simulation application and/or model. For example, lookahead information, which is crucial for a conservative protocol to be effective, is application-dependent [5]. If the choice is an optimistic protocol, there are several parallel simulation libraries intended for application programmers (e.g. GTW [6]). Even so, the programmer may still develop a runtime system, based on a general parallel library, because of optimizations specific to the application, or because the existing runtime does not suit the simulation's execution model. One example is the preliminary implementation of ModSim on the Time Warp Operating System (TWOS) [12]. The researchers eventually built their own runtime system, because TWOS presented an event-oriented, time-stamped message-based execution model which was difficult to match with Modsim's object-oriented, process-based model. It is therefore crucial that the chosen runtime have an efficient implementation, and present an appropriate interface to the target simulation.

In this paper, we selected four general parallel runtime libraries potentially relevant to simulation applications, and implemented the same conservative protocol and application benchmark on each of them. We study the four libraries in three main aspects: (a) programmability, i.e. how well the library interface supports the execution model; (b) performance, i.e. their relative execution times, and as compared to a sequential simulation; and (c) any additional mechanisms that allows the programmer to easily fine-tune the application performance.

The rest of the paper is organized as follows. Section 2 covers our application in more details, in particular the simulation protocol and simulation benchmark used. Section 3 summarizes the parallel runtime packages which we have chosen to study, and explains the rationale for our choices. In Section 4, we describe how we implemented our application for each package, and examine the fit between the library functionalities and simulation execution model. The performance results are presented in Section 4.2. We conclude our study and outline the future directions of our project in Section 5.

2. Parallel Discrete Event Simulation

On each of the four runtime systems, we implemented a parallel discrete event simulation (PDES) of a simplified but generic version of a manufacturing process, such as that found in semiconductor wafer manufacturing [4]. In a PDES, a physical process in the physical system is usually modeled by a *logical process* (LP) in the simulation model, and events in the physical system is simulated by communication between LPs using timestamped messages. Since LPs are executed in parallel, their simulation time may advance asynchronously. Thus, an LP may not always receive messages with increasing timestamps. However, in order to correctly simulate the physical system, each LP must process their incoming messages in their global timestamp order.

In the last decade, there are many protocols that have been proposed to solve the above synchronization problem. In general, these parallel synchronization protocols can be broadly classified into two approaches: *conservative* or *optimistic*.

In a conservative approach [3], an LP can make progress in simulation only when the causality is preserved. With this constraint, it is possible that a simulation

model may have deadlocks even if the simulated physical system does not. To avoid deadlocks, conservative protocols are usually required to send the updated local simulation time of an LP to other LPs using additional protocol messages. Handling protocol message is considered to be the major overhead in conservative approach.

In an optimistic approach [10], an LP is allowed to make progress as far forward in simulation time as possible. However, if there is a violation of causality discovered, an LP has to move backward in simulation time, rolling back its states and canceling the messages it had sent to other LPs. To process the rollback, an optimistic parallel simulation usually requires to periodically checkpoint the simulation state of an LP; to handle the cancellation of messages; and to carry out the rollback of state and redo the necessary computation. Therefore, the major overhead in optimistic approach is the processing of the rollback.

In this paper, we adopted a parallel simulation protocol that executes in lock-steps. The algorithm was first proposed in [2]. It is a conservative algorithm since at each lock-step an LP can only process the events that are safe to be simulated. We selected this algorithm in our study of runtime systems because it is simple to implement, compared to the optimistic protocols. In addition, its iterative nature also makes it easy to be implemented using both Cilk and BSP runtime systems.

2.1 Conservative Simulation Protocol

The algorithm is given in Figure 1. At each lock-step, the events whose simulation time is smaller than the SafeTime are processed. The calculation of SafeTime, which is set to be the maximum of InClock and the global simulation time (GST), uses both local and global information. Similar to the "null message" algorithm [3], a *link clock* is kept for each input link of an LP, which is the timestamp of the last message received on that input link[1]. InClock is defined as the minimum value of all input link clocks. Hence, no event can arrive on an input link with a timestamp that is smaller than the InClock value. GST is defined as the minimum of all local simulation times and the timestamps of all messages that have been sent but not yet received. Therefore, it is the smallest timestamp of any event in the whole system and so no event can be generated with a smaller timestemp than GST. The calculation of GST requires a global reduction operation, but InClock is calculated locally by each LP.

From the above discussion, it can be seen that our algorithm guarantees that there is no violation of causality. The major difference between our algorithm and other lock-step based parallel simulation algorithms, for example, YAWNS [11], is the calculation of the SafeTime. In [11], only the global information (i.e., GST) is used to determine up to what simulation time the current lock-step can simulate.

To show that deadlock cannot occur, we note that for each LP, we have SafeTime ≥ GST. Since GST is the smallest timestamp of any event in the whole system, at least those events with a timestamp equal to GST will be executed in the current lock-

[1] We assume that for each communication link, messages are received according to the order they were sent and that external events have a timestamp which is equal to the local simulation time of the sending LP. We also assume that there is no preemption of events simulated.

step. In each lock-step of the simulation, there must be at least one event with a timestamp equal to GST, so the simulation will progress and deadlock will not occur.

```
gst = 0;  OldBuff = BuffA;  NewBuff = BuffB; /* (1) global initialisation */

lp[i].st = 0;    lp[i].event_q = empty;    /* (2) local initialisation */
for j = 0 to lp[i].InNum-1 do    lp[i].clock[j] = 0;  BuffA[i,j] = empty;   BuffB[i,j] = empty;  end for
lp[i].state = InitialState();
for all InternalEvent ie caused by InitialState do
        OrderInsert(ie@OccurrenceTime(ie), lp[i].event_q);
end for
for all ExternalEvent ee by InitialState do
        OrderInsert(ee@lp[i].st, NewBuff[k,j])
                where ee is for LP k and LP i connected to jth input link of LP k;
end for

/* (3) main simulation loop */
while (gst <= endtime) do
        /* (4) swap buffers at start of a simulation lock-step (or cycle) */
        swap OldBuff and NewBuff;

        InClock = infinity          /* (5) calculate InClock */
        for j = 0 to lp[i].InNum-1 do
                if (OldBuff[i,j] != empty)  {
                        lp[i].clock[j] = LastElementTime(OldBuff[i,j]);
                        OrderMerge(OldBuff[i,j], lp[i].event_q);
                        OldBuff[i,j] = empty;  }
                InClock = min(InClock, lp[i].clock[j]);
        end for
        /* (6) calculate SafeTime */
        SafeTime = max(InClock, gst);   lp[i].out = infinity;

        /* (7) simulate all safe events */
        while (FirstElementTime(lp[i].event_q) <= SafeTime) do
                /* dequeue an event and process it */
                e = RemoveFirstEle(lp[i].event_q);
                lp[i].st = TimeStamp(e);    lp[i].state = Simulate(e);
                /* enqueue new internal events */
                for all InternalEvent ie caused by Simulate(e) do
                    OrderInsert(ie@OccurenceTime(ie), lp[i].event_q);
                end for
                /* output external events */
                for all ExternalEvent ee caused by Simulate(e) do
                    OrderInsert(ee@lp[i].st, NewBuff[k,j])
                        where ee is for LP k and LP i connected to jth input link of LP k;
                    LP[i].out = min(LP[i].out, TimeStamp(ee));
                end for
        end while

        /* (8) calculate smallest timestamp of any event in this LP */
        if (lp[i].event_q !=empty)  { SafeTime = FirstElementTime(lp[i].event_q);  }
        else  {  SafeTime = infinity; }
        SafeTime = min(lp[i].out, SafeTime);

        gst = min_reduce(SafeTime); /* (9) global reduction to calculate new gst*/
end while
```

Figure 1: Lock-step parallel simulation algorithm

In our algorithm, communication links between LPs are implemented using buffers. To avoid the conflicts in accessing the same buffer, two buffers are therefore used alternately in each lock-step for input and output, and pointers to these buffers are swapped in the main simulation loop at the start of each lock-step.

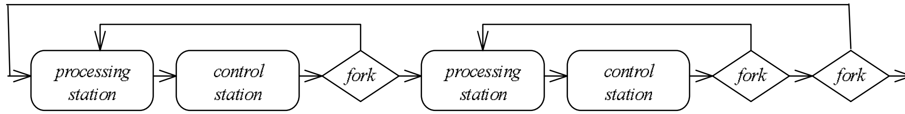


Figure 2: Production Line

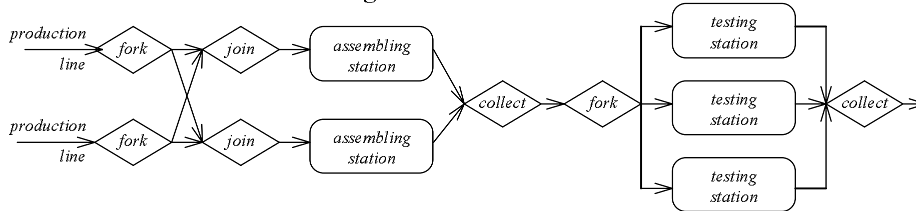


Figure 3: Assembling and Testing

2.2 Manufacturing Simulation Benchmark

Our simulation model of a generic manufacturing process consists of three stages: production, assembling and testing. There are a number of production lines, each producing a different component. Each production line is made up of a sequence of *processing* and *control stations*, as shown in
Figure 2. A processing station processes a part and sends it to a control station where quality control on the processed part is conducted. A re-work probability decides whether the part needs to be sent back to the processing station for re-processing.

An *assembling station* assembles a product using the components from a number of production lines. An assembled product will be tested by one of a number of *testing stations*.
Figure 3 shows the assembling and testing of products. As well as the four types of station *processing*, *control*, *assembling* and *testing*, the queueing network includes three other types of node:

- *fork*: Routes an input event to one of a number of output links according to the link probability,
- *join*: Generates an event on its output link when an input event is received on each of its inputs,
- *collect*: Generates an event on its output link when an input event is received on any of its inputs.

The simulation model is parameterized by the number of production lines, the number of processing stations per production line, the number of assembling stations and the number of testing stations. The first processing station of a production line also acts as a source with a mean arrival time for parts. Each of the four types of station has a mean service time. Note that *fork*, *join* and *collect* do not increment the simulation time.

3. Parallel Runtime Libraries

We compared an industry standard library (POSIX threads [8]) and three research-based libraries: Active threads [13], Cilk [1], and Oxford BSP [7] which is an implementation of Bulk Synchronous Parallel (BSP) model. Our choices were determined mainly by our target platforms which are machines support ingshared-address spaces, whether it is Non-Uniform Memory Access (NUMA) (e.g. SGI Origin 2000) or UMA (e.g. Sun Ultra2 Enterprise 3000).

Because multithreading is a common programming model on such platforms, three of our choices are thread-based libraries. They however offer different synchronization mechanisms and have different system performance.

- We included POSIX in our comparison, because it is an industry standard. If the POSIX overheads are reasonably low for an application, it can be advantageous to use POSIX because of its portability across different platforms.
- Active threads has a similar interface to POSIX threads, and is especially targeted for fine-grain irregular parallel computations. It also proivdes high-level abstractions for the programmer to take advantage of the memory hierarchy.
- Cilk has an efficient thread implementation, and a provably good scheduling and load balancing mechanism. Unlike the synchronization in POSIX and Active threads, a Cilk parent thread only sychronizes with its children by waiting for their termination.

Last but not least, we included Oxford BSP in our study, because it has a sychronous message-passing model, an alterntive different from the other thread-based libraries.

POSIX threads: The POSIX library provides the programmer with the means to create and synchronize threads. There are facilities for the programmer to influence low-level scheduling details, e.g. by specifying a thread's scope of contention (interprocess or intraprocess), and its scheduling priority. The defaults are taken care of by the operating system. POSIX 1003.4a [8] is an emerging industry standard for platform-independent thread interface, and is currently supported by different vendors. POSIX threads are light-weight, compared to process-based mechanims such as *fork*, but they are still too heavy-weight for fine-grain parallel programming.

We use POSIX as a baseline to study if the sychronization overheads of an industry-standard library are acceptable for a PDES application.

Active threads: Active threads [13] is a light-weight threads package targeted at irregular applications. Its API is similar to that of most thread packages. In addition to standard synchronization objects (e.g. mutexes and sychronizations), it provides a general mechanism for programmers to build data structure specific thread schedulers. Threads with similar properties (e.g. a similar scheduler) can be grouped together as a bundle. [13] reported that it outperforms vendor-provided thread package, and its context switch is only an order of magnitude more expensive than a function call. We derived our Active threads version by replacing a POSIX version's library calls by their corresponding Active thread API.

Cilk: Cilk [1] is a C-based language for multithreaded parallel programming. The Cilk runtime environment takes care of the low-level details like scheduling and

load balancing. The work stealing scheduler of the Cilk runtime system is particularly suitable for computations that unfold dynamically to give a hierarchical tree of Cilk procedures. In the implementation of our parallel simulation algorithm, LPs (or rather clusters of LPs) are spawned in each lock-step using a divide and conquer procedure, which passes the global simulation time to its children and thus to each LP.

Bulk Synchronous Parallelism (BSP): In the BSP model, computations proceed in terms of supersteps. A superstep consists of a computation phase, followed by a barrier synchronization during which global communication occurs. During the computation phase, each process computes independently, and can only access local data. BSP supports a Single Program Multiple Data (SPMD) programming model, and provides message-passing primitives to exchange data between processes during the global communication phase.

4. Comparisons

We compare the runtimes in terms of their (a) ease of programming (i.e. whether there is a good fit between the API and the application's execution model), and (b) performance, and (c) any additional mechanisms that allows the programmer to easily fine-tune the application performance.

4.1 Ease of Programming

To compare the implementations, we distinguish the following two abstractions: (1) the threads/processes in the libraries, and (2) the LP's in the simulation protocol.

The POSIX and Active threads versions share a similar computation structure (Figure 4). Each thread executes the simulation protocol for one or more LPs, each of which represents a simulation entity in the model. The threads synchronize at a barrier at the end of each lock-step, so that a new GST can be calculated. We can create more threads than the number of physical processors to take advantage of the library's default load-balancing mechanism. In the POSIX version, we expect to map multiple LPs to each thread because of its relatively high overheads. A similar strategy is used for the Active threads version because the number of LPs (usually in the range of 1000's) is greater than the number of threads (at most in the range of 100's) that can be efficiently supported in a system with processors in the range of 10's.

The Cilk version has a divide-and-conquer computation structure (Figure 5). Each thread executes the simulation protocol for one or more LPs *during one lock-step*. We have to create new threads at each lock-step, because threads synchronize only at termination. This results in more thread creation overhead, and accounts for its lower performance (Section 4.2).

```
Spawn threads and initialize // (1, 2) in Fig 1
while (GST <= endtime) {/* Main simulation loop – (3) in Fig 1, "master" thread */
        -- Swap buffer // (4) in Fig 1
        -- Activate all the threads
        -- Wait for all the threads to barrier-synchronize
        -- Compute new GST from shared array of SafeTimes  // (9) in Fig 1 }
-- Signal all the threads to terminate
```

```
/* Thread computation */
while (no termination signal) {
        -- Wait for activation
        for each LP e managed by thread {
            -- Compute InClock and SafeTime // (5, 6 ) in Fig 1
            -- Simulate all safe events for e (i.e. event time <= SafeTime)   // (7) in Fig 1
            -- Update e's new SafeTime in a shared array }  // (8) in Fig 1
        -- Barrier synchronize }
```

Figure 4: POSIX and Active threads version of simulation program

```
Spawn threads and initialize // (1, 2) in Fig 1
while (GST <= endtime) {   // Main simulation loop – (3) in Fig 1
        -- Swap buffer // (4) in Fig 1
        -- Spawn thread with current GST, and wait for thread to complete
        -- Update GST with new SafeTime result from thread }
/* Per-thread computation */
if (sufficient number of LPs) {
        for each LP e managed by thread {
            -- Compute InClock and SafeTime // (5, 6 ) in Fig 1
            -- Simulate all safe events for e (i.e. event time <= SafeTime) // (7) in Fig 1
            -- Compute LP's new SafeTime } // (8) in Fig 1
            -- Compute minimum of all LP's SafeTime // Part of (9) in Fig 1 }
} else {
        -- Split into two ranges for the LP's
        -- Spawn thread for each range, and with GST
        -- Wait two threads to complete
        -- Compute minimum of the two thread's SafeTime  }  // Part of (9) in Fig 1
Return new SafeTime
```

Figure 5: Cilk version of simulation program

```
Spawn threads and initialize // (1, 2) in Fig 1
/* Main simulation loop executed by every process.  A process has a buffer
   containing all messages sent to its LPs during the previous lock-step. – (3) in Fig 1 */
while (GST <= endtime) {
        -- Move event messages from buffer to each LP's input buffer
        -- Swap buffer // (4) in Fig 1
        for each LP  e managed by process {
            Compute InClock and SafeTime // (5, 6 ) in Fig 1
            -- Simulate all safe events for e (i.e. event time <= SafeTime) // (7) in Fig 1
            -- Compute LP's new SafeTime } // (8) in Fig 1
            -- Compute minimum of all LP's SafeTime
            -- Process 0 reduces all minimums, computes new GST and sends
                        update to all others  // (9) in Fig 1
        }
```

Figure 6: BSP version of simulation protocol

The BSP version is made complicated by the no-shared-data property of the BSP processes. Each lock-step in the protocol corresponds to a BSP superstep. At the barrier synchronization, the processes compute a new GST, and exchange messages containing new events. Our implementation provides a buffer for each process, to hold all the input events for its LP's. An event sent by an LP 1 in process A to another LP 2 in process B, is first put in B's buffer. The event is then moved to LP 2's input event buffer at the start of next lock-step (Figure 6). If LP 1 and LP 2 are in the same process, the event is just inserted into LP 2's input event buffer. (Note that

the events in each LP's input buffer are moved to its event queue in the simulation algorithm.)

The Oxford BSP library's performance degrades when the number of processes is greater than the number of physical processors. Due to this limitation, we cannot create more processes than processors, to make use of the operating system's default load-balancing mechanism.

None of the four runtime systems allows an efficient one-to-one mapping between an abstract LP and a thread (flow of control). But for POSIX and Active threads, a many-to-one mapping can be easily implemented. In addition to the many-to-one mapping, in Cilk, the creation of threads is tied to the lock-step notion from the protocol. But on the other hand, Cilk's thread creation construct is straight-forward and well-suited to tree-like computations. In the BSP version, for efficiency reason, each processor runs exactly one process, and platform-independence is reduced. This is implementation-specific, and subjected to modification in future. BSP would have been easier to use for distributed platform. From the implementations, it seems that Cilk and BSP will be difficult to use (though possible) for simulation protocols which are not globally synchronous (e.g. time warp).

4.2 Performance

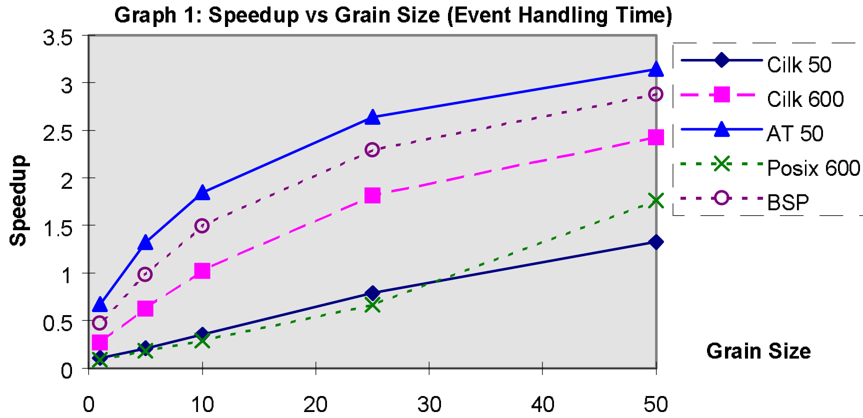
We focused on studying the following performance questions: (1) what LP granularity will sufficiently mask the overheads to give an efficient run, (2) given a good granularity, how do the various versions scale with varying number of processors.

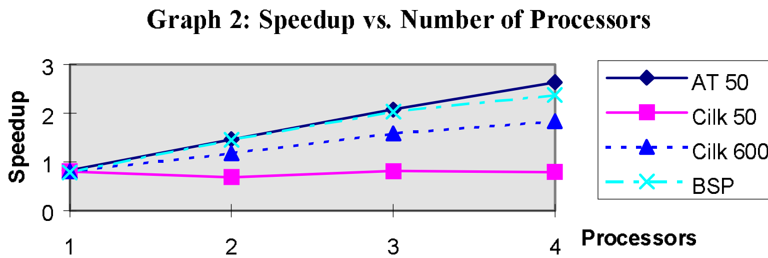
In all our implementations, multiple LP's are mapped to a single thread. To study (1), there are therefore two parameters we can vary: the *event-handling time* of each LP and the number of LP's per thread (which we refer to as *threshold*). We tried five values for the event-handling time: 1, 5, 10, 25, and 50 µs, and three threshold values: 50, 300 and 600. We observe that for Cilk, given an event-handling time, its performance varies a lot for different thresholds. For comparison, we selected the largest and smallest threshold (i.e., Cilk 50 and 600). Active thread timings show little variation for different thresholds, so we selected the smallest threshold (i.e., AT 50), to study its performance for small-grain threads. POSIX timings also show little variation, but we selected the largest threshold (i.e., Posix 600) because POSIX threads are known to be relatively coarse-grained. BSP has one process per processor, so we simply distribute the LPs evenly among the processes.

For comparison purpose, a sequential simulator was also implemented using the same event queue management routines as that used in the parallel implementation. We compare the parallel timings to the corresponding sequential timings for speedups (Graph 1). As expected, the speedups improve with larger event handling time. We pick 25 µs for our later experiments, because it is reasonably small to be used to represent fine-grain event-handling.

Because POSIX does not show any speedup at grain size 25 µs, Graph 2 only shows the speedup curves of the three research libraries. We see that the speedup curve for Cilk is quite sensitive to the thread granularity (Cilk 50 vs. Cilk 600). Both BSP and Active threads perform quite well in general. The difference is that Active threads continue to show speedup for small threads, while BSP has one process per

processor, and has large threads. From Graph 2, we can also see that the performance of Active threads is also more scaleable than Cilk and BSP. With increasing number of processors, it is likely that Active threads will perform much better than Cilk and BSP. Results on an 8-CPU Ultra1 will be included in the final version of this paper.





The three versions based on research libraries all offer better performance than POSIX, and perform well with the correct thread granularity. For small thread granularity, only Active threads managed to show speedup.

4.3 Mechanisms for Performance Optimization

Because memory accesses is a major potential source of performance bottleneck on current platforms, it is crucial to provide parallel programmers with the mechanisms to improve their program memory's spatial and temporal locality, and/or to implement their own memory management scheme.

In our protocol, one simple way to improve memory locality is to associate a pool of events for each LP or each thread. New events are allocated from the pool, and finished events are returned to the pool. The sequential timing in Section 4.2 makes use of this memory pool mechanism.

Among the three research libraries, our experience was as follows: (1) For the Active threads version, we implemented a per-LP memory pool. This memory management scheme is more efficient than the UNIX library because it improves spatial locality. (2) Cilk cannot make good use of per-thread memory pool due to the short-lived nature of each thread. (3) For BSP, we tried to implement a per-process memory pool, but did not succeed. We suspect that it was because the system

memory routines interfered with BSP's memory system which enforces memory protection between different processes.

Cilk automatically takes care of thread scheduling and load balancing. This is an advantage because it makes the system transparent. But it is also a disadvantage, because there is no way for the programmer to provide additional information, for the runtime to do locality-based scheduling.

BSP automatically enforces some extent of spatial locality, simply from its programming model. Active threads provide facilities for programmers to bundle threads with a scheduler to improve locality and fine-tune performance, but we did not use this in our implementation because the Active threads version was quite efficient, and such additional fine-tuning was outside the scope of this study.

5. Conclusion

This paper studies the implementation of a lock-step conservative simulation protocol using four general parallel runtime libraries. A simulation entity in our manufacturing model corresponds to an LP in the simulation protocol. For all four libraries, multiple LP's are handled by each thread / process.

The protocol can be easily implemented using any of the multithreaded libraries (POSIX, Active threads and Cilk). The main distinction is that in the POSIX/Active threads model, each thread manages all the simulation entities (LPs) throughout their lifetime, while a Cilk thread manages its LPs during a single lock-step. The BSP version is more complex because we have to implement an additional buffer for each process to store its LPs' input events. Such a buffer is neither part of the protocol, nor the simulation model, and distracts from the programming task. This complexity would not arise if we can efficiently map an LP to a process. We also note that lock-step nature of the protocol maps well onto Cilk and BSP, but a non lock-step algorithm may not be as straight-forward.

We obtained our timings on an 4-CPU Ultra2, but will also include 8-CPU Ultra1 timings in the final version of this paper to compare the scaling with respect to new architectures. In general, the POSIX threads' overheads are still too high for fine-grain computation. The research libraries all show speedups on 4 processors, but to different extents. Cilk speedup curves improve with larger thread granularity, while Active threads' speedup is consistent across different thread granularities. Such a comparison is of course only a snapshot of the existing libraries, but we can in future reuse our application for new implementations and libraries to study any new functionalities or performance improvement.

Our comparison study used a simple manufacturing process model. We are currently integrating the various aspects of a virtual factory model, including business processes, manufacturing and communications network support, and evaluating the different protocols to find out which will give better performance for our application. We also foresee the need to study load balancing and scheduling issues when the implementation work on the prototype gets underway.

Acknowledgments: We would like to thank Sanjay Jain (Gintic, Singapore), and Boris Vaysman (International Computer Science Institute, Berkeley, USA) for their comments on preliminary drafts of this paper, and their various suggestions in this study. This research is supported by National Science and Technology Board,

Singapore, under project: *Parallel And Distributed Simulation of Virtual Factory Implementation*.

References

1. R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, Vol.37, No.1, pp.55-69, 1996.
2. Wentong Cai, E. Letertre and S. J. Turner. Dag Consistent parallel Simulation: a Predictable and Robust Conservative Algorithm. In *Proceedings of 11th Workshop on Parallel and Distributed Simulation (PADS'97)*, pp. 178-181, Lockenhaus Austria, June 1997.
3. K. Mani Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Trans. On Software Engineering*, Vol.SE-5, No.5, pp.440-452, Sept. 1979.
4. G. Feigin, J. Fowler and R. Leachman. Semiconductor Wafer Manufacturing Data Format Specification. SEMATECH Technical Report, July 1994.
5. Richard M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, October 1990, Vol. 33, No. 10, pp. 31 – 53.
6. Richard M. Fujimoto, Samir R. Das, Kiran S. Panesar, Maria Hybinette and Chris Carothers. Georgia Tech Time Warp (GTW Version 3.1) Programmer's Manual for Distributed Network of Workstations. GIT-CC-97-18. July 3, 1997. College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280.
7. Jonathan M.D. Hill. Installation and User Guide for the Oxford BSP toolset (v1.1) implementation of BSPlib. Oxford Parallel Computing Laboratory, Oxford University. June 1997.
8. The Institute of Electrical and Electronics Engineers. Portable Operating System Interface (POSIX) -Part 1: Amendment 2: Threads Extensions [C Language]. *POSIX P1003.4a/D7.* April 1993.
9. Sanjay Jain. Virtual Factory Framework: A Key Enabler for Agile Manufacturing. *Proceedings of 1995 INRIA/ IEEE Symposium on Emerging Technologies and Factory Automation*, Paris, Oct. 1995, Vol. 1, pp. 247-258, IEEE Computer Society Press, Los Alamitos, CA.
10. David R. Jefferson. Virtual Time. *ACM Trans. On Programming Languages and Systems*, Vol.7, No.3, pp.404-425, July 1985.
11. David M. Nicol, "The Cost of Conservative Synchronization in Parallel Discrete Event Simulation", *Journal of the ACM*, Vol.40, No.2, pp.304-333, April 1993.
12. David O. Rich and Randy E. Michelsen. "An Assessment of the ModSim/TWOS Parallel Simulation Environment" Proceedings of 1991 Winter Simulation Conference, pp.508-518, 1991.
13. Boris Weissman. Active threads manaul. International Computer Science Institute, Berkeley, CA 94704. Technical Report TR-97-037. 1997.