To appear in the proceedings of the *International Conference on Parallel Processing, 1995*.

Portable Runtime Support for Asynchronous Simulation *

Chih-Po Wen and Katherine Yelick
Computer Science Division, University of California
Berkeley, CA 94720
cpwen@cs.berkeley.edu yelick@cs.berkeley.edu

Abstract

We present library and runtime support for portable asynchronous applications, using event-driven simulation as an example. Although event-driven simulation has a natural source of parallelism between the simulated entities, real speedups have been hard to obtain because of the fine-grained, unpredictable communication patterns. Language and systems software support is also lacking for asynchronous problems. Our runtime supports makes the applications portable, eases performance tuning, and allows code re-use between applications. Our goal is to support a range of platforms with varying performance characteristics, from special-purpose multiprocessors through networks of workstations. We discuss the performance issues in the runtime support, describe a distributed event graph data structure, and present performance numbers from a parallelized timing simulator called SWEC.

1 Introduction

Parallel applications can be classified according to their degree of irregularity. Fox identifies three classes: *synchronous* algorithms have little or no data-dependent behavior and fit into an SIMD execution model; *loosely synchronous* problems are spatially irregular but temporally regular and can be expressed in SPMD languages with alternating communication and computation phases; *asynchronous* problems are irregular in time and space and are the most difficult to parallelize [Fox91]. Data parallel languages like HPF [Hig93] and NESL [Ble93] are well-suited to synchronous problems, and with sufficient compiler [BCF+93, BC90] and run-time support (e.g., the PARTI system [SBW91]), can be used for loosely synchronous problems. In this paper we describe library and runtime support for asynchronous applications.

The goal of our research is to provide software systems to make asynchronous applications easier to parallelize, using extensive runtime support and a distributed data structure library. Portability is a primary goal. The programs developed using our system run and exhibit good performance on a range of machines. In this paper, we use a conservative parallel version of the SWEC simulator, an asynchronous event-driven simulator, as a running example to study programmability, performance, and portability issues of our system. The main distributed data structure in this problem is an *event-graph*, a component of our Multipol library [CDI+95]. In previous work, we described a parallel implementation of the SWEC simulator that used optimistic concurrency [WY93]. The implementation performed well on the CM5 multiprocessor, but contained machine-specific techniques and was not organized to allow for code re-use in other simulators. This work addresses those concerns.

Our research targets distributed memory architectures such as the CM5, Paragon, SP1/SP2, and networks of workstations. A common characteristic of distributed memory machines is that the overhead and latency of accessing remote data is much larger than accessing local data, and the bandwidth of remote access improves with the size of the data. Solving irregular problems on these machines is particularly challenging, because communication is fine-grained and unpredictable, leaving few opportunities for compile-time optimization or runtime preprocessing.

Our runtime system employs two techniques to

*This work was supported in part by the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026, by the Department of Energy grant DE-FG03-94ER25206, and by the National Science Foundation (number CCR-9210260 and number CDA-8722788). The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

reduce communication costs: *split-phase operations* built from atomic threads, which hide the latency of remote operations, and *message aggregation*, which reduces the overhead of communication. On top of this, the Multipol data structures provide reusable abstractions for irregular structures. One of these, an *event graph*, provides order-preserving, flow-controlled message delivery between a set of logical processes. It also takes advantage of the process graph structure to optimize for communication efficiency.

The rest of the paper is organized as follows. Section 2 gives an overview of the application, which is a timing simulator called SWEC. Section 3 describes our runtime support, drawing examples from SWEC to motivate the design. Section 4 describes the interface and semantics of the event graph and sketches the algorithm used in the parallelized SWEC. Section 5 gives the performance results and uses statistics from our experiments to evaluate the effectiveness of our runtime system. Section 6 describes previous work on portable software support and discrete event simulation. Section 7 concludes the paper.

2 Overview of SWEC

The SWEC program is mainly used to perform timing simulation for digital MOS circuits [LKMS91]. It partitions a circuit into loosely coupled subcircuits (as shown in Figure 1), each of which can be simulated independently within a time step. At the end of a time step, if the subcircuit's new state cannot be extrapolated linearly from the old state within some error margin, the new state is propagated to the fanout subcircuits. This communication is referred to as an event. Events occur at unpredicatable times and vary in frequency depending on the non-linearity of the voltage. The time step size used by the subcircuits depend on their states. SWEC uses smaller timesteps for subcircuits with more activity to improve accuracy, and larger timesteps for subcircuits with little activity to save computation. There are no global synchronization points, since each subcircuit is simulated with a variable number of timesteps that cannot be predicted in advanced.

In the sequential implementation, the subcircuits are always scheduled in strict ascending time order, so that all relevant events are processed before any affected subcircuit computation takes place. Figure 2 sketches the sequential algorithm. A subcircuit

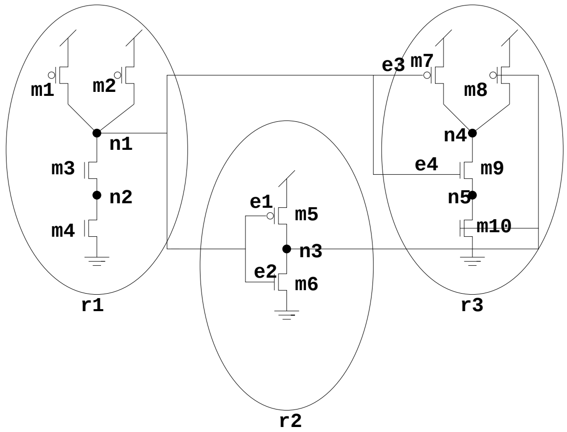

Figure 1: Partitioning the circuit into subcircuits. The subcircuits (denoted by `r`) contains MOS transistors (denoted by `m`) and voltage points (denoted by `n`). The transistors are connected to their fanin voltage points via the gates (denoted by `e`).

`s` has two attributes: `s.time`, the current simulation time of the subcircuit and `s.step` the current duration of a time step. Subcircuits are placed in a priority queue, `p`, ordered by the estimated next time point (`s.time + s.step`).

The priority queue in SWEC offers little room for parallelization, because it imposes a total order on the scheduling of subcircuits. Our parallel implementation distributes the subcircuits among the processors, and the simulation proceeds in a data-flow manner. The parallel algorithm is a typical case of distributed asynchronous simulation, where each subcircuit corresponds to a logical process, and each event corresponds to a collection of messages sent to the fanout subcircuits. We adopt the conservative approach to parallel asynchronous simulation [KC81], scheduling a subcircuit only when all input events have been processed. Previously, we showed that conservative parallel simulation is primarily useful for combinational circuits (circuits without feedback paths); optimistic scheduling is much more effective for sequential circuits[WY93]. We are developing an optimistic analog to the conservative Parallel SWEC, but in this paper we focus on the conservative example.

Parallel SWEC is irregular in computation and communication. The number of time steps as well as their costs depend on the state of simulation, which is not known in advance. The number of messages is also data dependent, so their storage cannot be preallocated. Furthermore, the messages, usually small,

```
For all subcircuits s,
  initialize s.time and s.step
  insert s into p using key (s.time + s.step)

While min key of p < end of simulation,
  delete s with min key from p
  simulate s from s.time up to (s.time + s.step) and
      update its state
  for each voltage point v of s,
    if new state of v cannot be linearly extrapolated
        from previous state
      for each fanout subcircuit s' of s at v
        update the state of the mosfet driven by v in s'
        reduce s'.step if necessary
        delete s' from p
        insert s' in p using key (s'.time + s'.step)
  s.time = s.time + s.step
  s.step = projected time step based on new state of s
  insert s into p using key (s.time + s.step)
```

Figure 2: The sequential SWEC algorithm.

are sent at unpredictable times. To resolve load imbalance caused by computation irregularities, we use simple static load balancing heuristics for distributing the subcircuits across processors. Our heuristics work well in practice and avoid the communication overhead of dynamic load balancing. In the remainder of this paper, we focus on communication irregularities and how small asynchronous messages are handle in a portable manner.

3 Runtime Support

To achieve good performance on distributed memory machines, three communication costs must be addressed: overhead, latency, and bandwidth. In this section, we explore these issues and present the solutions used in our runtime layer.

3.1 Overhead Reduction

Communication overhead refers to the processor time spent in setting up the communication and injecting data into the network. It can be decomposed into the fixed start-up overhead for setting up the communication, such as allocating storage or performing kernel calls, and the overhead per byte for injecting the message. Even for machines with like the CM5 with small hardware messages and lightweight communication such as Active Messages [vECGS92], the startup cost observed by irregular applications may be significant. Active messages avoid storage allocation by using fixed hardware packet sizes and by requiring that the user provide an address range in memory for the message data to be written. Structuring the program based on a fixed packet size limits portability, and for asynchronous programs, the cost of buffer management may resurface in the users' code.

Our runtime system accumulates small, asynchronous messages in a continuous buffer until its size exceeds a certain threshold, or until no other computational threads are eligible for execution. The aggregated messages in the buffer are then sent as a single message using a bulk communication primitive. Message aggregation, a technique that is well-known for bulk-synchronous algorithms and libraries [BSS91, KB94], reduces communication overhead by amortizing the start-up overhead over many messages.

Although message aggregation reduces the amortized communication start-up, it incurs the overhead for data copying, and thus increases the overhead per byte. It may also increases the observed latency of remote operations, since messages may be delayed in the aggregation buffer. The exact tradeoff depends on the application, the machine architecture, as well as the input. Therefore, we expose the degree of aggregation to the programmer for performance tuning, but ensure that all message will eventually be delivered.

3.2 Latency Hiding

Irregular applications typically have dynamic data structures such as graphs, trees, and tables. Rather than simple fetch and store operations, as one would have on arrays or static graphs, dynamic structures have operations to add or delete nodes from linked structures as well as non-trivial operations to observe the state of the structure. For such data structures, there can be many sources of latency in an operation: the delay due to message aggregation, the network latency, the delay due to scheduling at the remote processor, the computation time at the remote processor, and the latency of sending back the result. Since the total latency can be long, support for multithreading is required to overlap the latency with other computation.

Our runtime system supports split-phase operations implemented as user-level *atomic threads* for latency hiding. A atomic thread is a finite computation that is guaranteed to run atomically. A split-phase operation is built from two or more such threads. High-level synchronization constructs such as suspension are implemented on top of atomic threads using continuations. Atomicity eases programming by eliminating locking for simple read-modify-write operations. Unlike system level threads, the context of our threads are explicitly managed by the programmer, so only the required variables are passed from one thread to its continuation. The threads synchronize using counters, similar to those used in Split-C [CDG+93]. Every split-phase operation takes a counter as input argument, which is incremented when the operation completes. A separate continuation thread can be created to wait on the value of the counter. The counter automatically enables the continuation thread for execution when the operation completes, without requiring polling by the programmer.

The runtime system also provides the programmer with mechanisms for building customized schedulers. For example, a scheduler which has knowledge of the simulation time is used to schedule the subcircuit threads in parallel SWEC. The user schedulers are fairly invoked by the system, so the scheduling policy of one data structure can be fine-tuned for performance without introducing unexpected deadlocks.

3.3 Bandwidth Optimization

Using message aggregation, the overhead per byte, or the communication bandwidth, becomes the determining factor of communication efficiency. Communication bandwidth can be improved by implementing bulk communication with nonblocking primitives. The runtime system interfaces with all machine architectures using the *nonblocking store* primitive, which transfers a block of data from a local address to a remote address, increments a local counter when the local buffer can be reused, and invokes a remote thread when the transfer is complete. The nonblocking store primitive saves the copying cost for queuing messages at the sending processor, and may improve network utilization by interleaving the injection of packets from multiple messages that are to be sent to different processors.

3.4 Summary of Runtime Support

The runtime system supports a small set of primitives for managing atomic threads and three forms of split-phase communication: *put*, which transfers a block of data from a local address to a remote address, *get*, which transfers a block of data from a remote address to a local address, and *remote thread invocation*, which creates a thread on a remote processor. The remote threads are like any local threads and may perform arbitrary computation. They are different from the network handlers, which the runtime system uses in a restricted manner to drain data from the network as fast as possible. The put and get operations are used for bulk, regular communication, while remote threads are usually used for irregular communication such as sending events in SWEC.

4 A Distributed Event Graph Data Structure

The dominant communication in asynchronous simulation is the propagation of events. Storage must be allocated for the event messages before they are sent, and the messages must arrive in their time stamp order for the conservative method to work. In this section, we introduce a data structure called an event graph, which supports in-order, flow-controlled message delivery in a static network of nodes. The event graph also takes advantage of the graph structure to optimize for locality.

4.1 Interface and Semantics

The event graph can be thought of as a graph with fixed-sized FIFO buffers attached to the edges, with which the nodes send and receive events. We refer to the size of these FIFO buffers as the *edge capacity.* There are five operations on the event graph, all of them split-phase:

- `MakeEventGraph`: create a distributed event graph on all processors based on an input graph. The programmer can write custom partitioners to assign nodes to processors. The programmer also specifies the edge capacity. `MakeEventGraph` completes when the event graph is ready for use on all processors.
- `SendEvent`: propagate an event from a node to all its fanout nodes. `SendEvent` completes when storage for the messages has been allocated in all fanout edges and the event has been copied. Its completion does not guarantee that the event messages are immediately available, although it is guaranteed to arrive in finite time.
- `ReceiveEvent`: read and/or remove an event message from an incoming edge of a node. The removal frees up space for more incoming messages. The programmer has the option

to wait for new messages when the edge contains no message. The programmer can also query the status of each edge, and uses non-split-phase versions of `ReceiveEvent` for better performance.

- `WaitForEvent`: wait until some event message is available for a node to receive.

- `Freeze/UnFreeze`: freeze and unfreeze the state of the event graph for taking *distributed snapshots*. `Freeze` suspends all split-phase mutators, such as `SendEvent`. When `Freeze` completes, the events are available using `ReceiveEvent`. No event can be "stuck" in the communication layer or the network. `Unfreeze` re-enables all mutators. The `Freeze/Unfreeze` operations are useful for detecting global properties in applications using distributed data structures. For example, they are used in parallel SWEC to detect deadlocks. They are also used to compute the global time in an optimistic parallelization of SWEC (currently under development).

Two operations on a data structure are said to *interfere* if their behavior under concurrent execution is undefined. For the event graph, `Freeze` interferes with all mutator operations that can temporarily leave the data structure in an inconsistent state. Some of the `ReceiveEvent` operations that modify the edge buffers are not split-phase, and as a result they interfere with `Freeze`. The programmer must insert sufficient synchronization in the program to ensure these operations are not issued when `Freeze` is in progress. Also, `SendEvent` interferes with `SendEvent` on the same node, since the meaning of concurrent enqueues to the same FIFO is not clear.

4.2 Implementation Techniques

The implementation of event graph uses the following techniques to improve performance:

- Lazy evaluation of `SendEvent`. We weaken the semantics of `SendEvent` so that its completion guarantees the event will arrive in finite time, but does not guarantee it is immediately available. The semantics requires no acknowledgement from the remote processor. The programmer can use the `Freeze` operation to ensure that all events are globally visible.

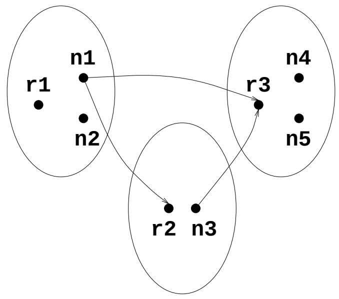

Figure 3: Mapping of circuit to event graph. The circuit shown in the previous example is mapped to an event graph. The nodes derived from the same subcircuit are allocated on the same processor, so that the simulation steps can be performed using local data.

- Caching event messages. The edges are allocated on the receiving processors, and all `ReceiveEvent` operations are handled locally. To further reduce communication, if an event is sent to two different node on the same processor, its value is cached on the remote processor. The caching is static because the graph structure remains fixed, so the storage for caching can be pre-allocated.

- Replicating the buffer control state. Sending an event requires that space be available on all fanout edges. We replicate the flow-control state of the edges (such as the number of elements) on the sending processor, and update the replica lazily so that the flow-control overhead can be amortized over many operations. When the edge capacity is large, most `SendEvent` operations can usually proceed without requiring communication for buffer space allocation.

4.3 Parallelizing SWEC using Event Graph

We parallelize SWEC using the event graph as follows. One distributed event graph is created for the entire circuit to handle all messages. Each subcircuit is a node in the graph for receiving event messages from its fanin subcircuits. Each voltage point is also a node for propagating its state. The mapping of circuits to event graph is illustrated in Figure 3. Evidence of the power of distributed data structures comes from our programming experience. The event-graph took a couple weeks to design, debug, and optimize. Given this, and a familiarity with

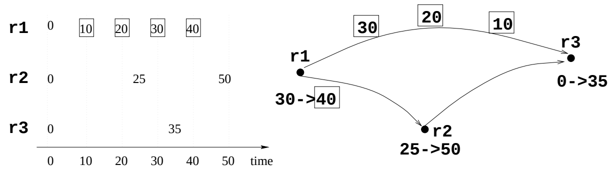

Figure 4: Deadlock due to storage constraints.

SWEC, the parallel conservative simulator took only a few days to implement.

Since we target combinational circuits, the data dependence graph of the circuit is acyclic, and deadlock due to the lack of global information cannot occur. However, because the event graph places an upper bound on the number of outstanding messages, the subcircuits sending new events may have to wait for its fanout subcircuits to release buffer space. This creates cycles in the overall dependence graph. Therefore, deadlock due to storage constraints can still occur. The problem is illustrated in Figure 4. The left-hand picture shows a sequential simulation with numbers denoting simulation steps and boxed numbers denoting steps that produced an event. In the parallel version on the right, the edge capacity is three events. The parallel simulation is deadlocked because `r2` has not produced an event to tell `r3` that it can proceed beyond time zero. Therefore, `r1` blocks because event 40 cannot be propagated due to the lack of space and `r2` blocks waiting for `r1`. If buffer space were unbounded, `r1` and `r2` would continue running, eventually producing an event from `r2` that would allow `r3` to proceed.

We resolve deadlocks using a combination of null messages and deadlock detection and recovery. A null message is an event that carries only the simulation time. A subcircuit sends a null message if it proceeds for several time steps and blocks without producing any event message. Our heuristics produce enough null messages to eliminate deadlocks in practice.

5 Performance Results

In this section, we report on the performance of the parallel timing simulator on various platforms, and provide statistics to show the effectiveness of our approach.

We have ported the runtime system to the CM5, SP1/SP2 and Paragon. The CM5 port is built on the Thinking Machines active message layer, called `CMAML`, the SP port uses the IBM message passing

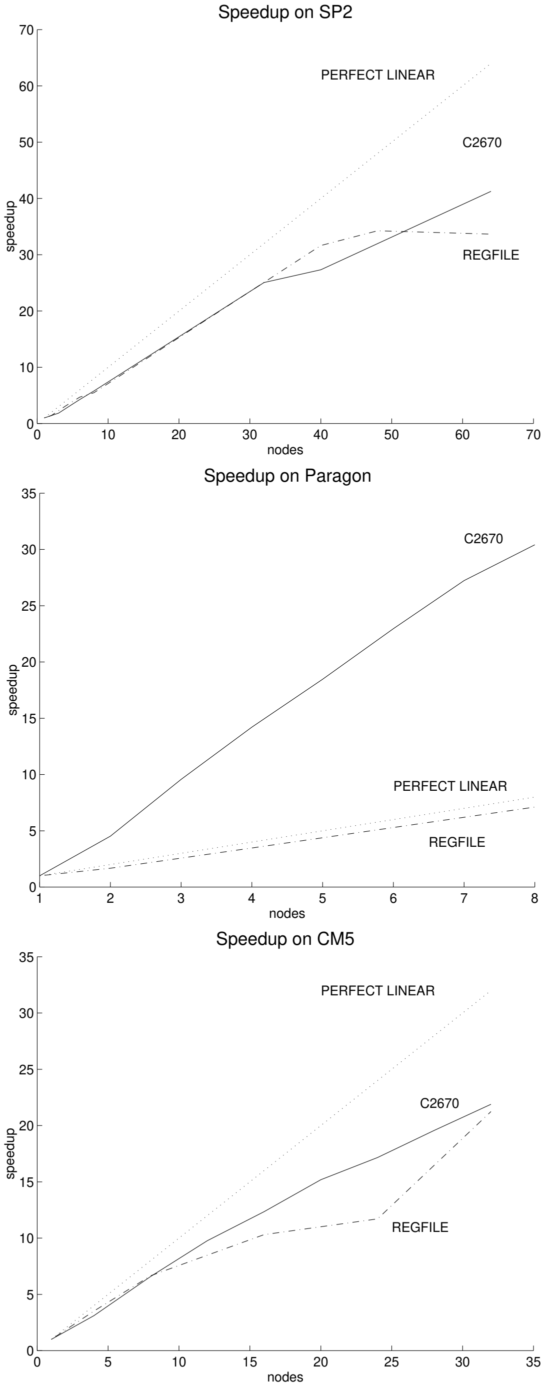

Figure 5: Speedups of the conservative parallel SWEC. The speedups are with respect to the 1 processor parallel implementation.

	subcircuits	mosfets	time steps	events	SP2	Paragon	CM5
C2670	2033	5364	2510047	760166	402 (243)	2159 (3046.4)	3916 (84 0.5)
REGFILE	325	4832	177987	957412	112.4 (153.7)	324.7 (460.3)	622 .7 (518.48)

Table 1: Statistics from the benchmark circuits. The execution times of the sequential SWEC on a single processor of machine is shown, with the running time of the one-processor parallel code in parentheses. All times are in seconds.

library `MPL`, and the paragon port is based on the `NX` communication library. We use the active message developed at Berkeley for both the SP and the NX ports [Lun94]. The SP and Paragon have higher bandwidth networks and faster processors than CM5. In general, the gap between the computation and communication performance is more pronounced for the SP and Paragon than for the CM5.

Table 1 describes the benchmark circuits. Due to differences in float-point characteristics, the number of time steps performed and consequently the number of events are slightly different among the three machines. REGFILE is a 32-bit register file, and C2670 is an unknown circuit from the ISCAS benchmark suite. Both circuits are combinational. C2670 has quite uniform computation granularities for the subcircuits, which usually contain only a few transistors. The computation of REGFILE is dominated by 32 very large subcircuits which represent the 32 bit slices of the register file. Notice that parallel SWEC on 1 node sometimes outperforms the sequential SWEC. Although parallelization causes communication and threading overhead, the distributed, data-driven approach of parallel SWEC alleviates the bottleneck of a centralized priority queue in SWEC, whose accessing cost grows with the number of subcircuits.

The speedups of our parallel implementation is show in Figure 5.[1] The curves are nearly linear for C2670, indicating good scalability for larger circuits. Both C2670 and REGFILE show speedup of over 5 on the 8-node Paragon, over 20 on the 32-node CM5, and over 41 and 33 on the 64-node SP2. The superlinear speedup of C2670 on the Paragon will be explained later.

Two program parameters can be set to fit the machine characteristics for better performance: degree of message aggregation and edge capacity. Figure 6 shows the impact of these two parameters on performance. The running times in the graphs are normalized with respect to the minimum time in each curve to show the percentage of increase in time.

Message aggregation is shown to be essential for performance. For C2670, the running time can increase by more than a factor of 3 when aggregation is not performed. The effect of aggregation is not as significant for REGFILE, which has very large subcircuits so that the communication time occupies a smaller portion of the execution time. In most cases, the running time increases when the degree of aggregation exceeds a certain threshold. This is because aggregation delays the progress of the simulation, so it becomes counterproductive unless there is sufficient parallelism. The threshold is about 2K bytes for C2670.

The effect of edge capacity is dependent on the input as well as the machine configuration. A large capacity improves performance by increasing concurrency, and consequently reduces the number of null messages that have to be sent to avoid deadlock. The increase in concurrency also provides more threads for hiding latency, as well as more opportunities for message aggregation. The results show that C2670 is less sensitive to edge capacity than REGFILE, because it has more parallelism (more subcircuits). The increase in running time for REGFILE can be as much as 60% on the SP1 and Paragon when the edge capacity is too small.

A large edge capacity, however, may degrade the performance of the memory hierarchy because it requires more memory. This is demonstrated by the running time of C2670 on Paragon, which increases by a factor of 3 when the edge capacity is too large. The simulation of C2670 may require more than 300M bytes of memory, and our local Paragon configuration provides only about 10M bytes of physical memory per node. This also explains the superlinear speedup within the parallel implementation for C2670 in Figure 5, since for a smaller number of processors the machine may spend a significant amount of time paging.

[1] We were not able to obtain some of the SP2 results. For processor numbers 1 through 8, an SP1 was used with performances scaled to reflect the speed difference.

6 Related Work

Parallel asynchronous simulation is a well studied problem. Chandy and Misra developed the conservative method[KC81]. Jefferson introduced the optimistic method, also known as the time-warp algorithm[Jef85]. In our prior work[WY93], we compared the potential of the both methods for parallelizing SWEC[LKMS91], and developed a CM5 specific implementation using the optimistic method.

Recent research has produced a variety of runtime support such as the Chare kernel [SK91], Nexus[FKOT91], and the compiler-controlled threaded abstract machine (TAM) [CSS+91]. Nexus provides mechanisms similar to remote thread invocation, but is more heavyweight than our runtime system, because it supports arbitrary thread suspension and heterogeneous computing. The threads in our runtime system are similar in spirit to the TAM threads, but do not require compiler support for static thread allocation, and are intended for coarser grained, dynamically generated threads.

The idea of aggregating small messages to reduce communication overhead can be found in the Fortran-D compiler[HKT91], which uses message vectorization in parallel loops to reduce overhead, and in PARTI[BSS91], which uses runtime preprocessing to pre-allocate storage for aggregated array accesses in bulk synchronous programs. It is a common technique for regular, array-based computations and for data structures that are irregular in time space but not time.

There have been a number of attempts at developing application-specific distributed data structures such as irregular grids[BSS91, KB94] and oct-trees[WS93]. As with aggregation, these data structures are targeted toward loosely synchronous, rather than asynchronous applications.

7 Conclusions

We have presented a runtime communication layer for a general class of irregular problems, and a distributed even graph data structure for asynchronous simulation. The runtime layer provides and portability layer for the Multipol library. Performance portability is obtained using split-phase operations, multithreading, and message aggregation. We demonstrated good performance on two nontrivial circuits. We also quantified some of the performance trade-offs that make portability difficult. The combination of runtime support and distributed data structures were shown to be effective for writing portable parallel programs for asynchronous simulation. Future work includes the development of an optimistic parallel implementation using the infrastructure described in this paper, application of these data structures to other problems, and the automatic tuning of program and data structure parameters for improving performance.

References

[BC90] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. In *Frontiers '90*, pages 471–480, College Park, MD, October 1990.

[BCF+93] Zeki Bozkus, Alok Choudhary, Geoffrey Fox, Tomasz Haupt, and Sanjay Ranka. A compilation approach for Fortran 90D/HPF compilers on distributed memory MIMD computers. In *Languages and Compilers for Parallel Computing*, pages h1–h23, Portland, OR, August 1993.

[Ble93] Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, CMU School of Computer Science, Pittsburgh, PA, April 1993. Updated version of CMU-CS-92-103, January 1992.

[BSS91] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory multiprocessors. *Concurrenty: Practice and Experience*, pages 159–178, June 1991.

[CDG+93] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Supercomputing '93*, Portland, Oregon, November 1993.

[CDI+95] Soumen Chakrabarti, Etienne Deprit, Eun-Jin Im, Jeff Jones, Arvind Krishnamurthy, Chih-Po Wen, and Katherine Yelick. Multipol: A distributed data structure library. Technical Report To appear, University of California at Berkeley, Computer Science Division, 1995.

[CSS+91] D. Culler, A. Sah, K. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *Proc. of 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Santa-Clara, CA, April 1991.

[FKOT91] Ian Foster, Carl Kesselman, Robert Olson, and Steve Tuccke. Nexus: An interoperability toolkit for parallel and distributed computer systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, 1991.

[Fox91] G. C. Fox. The architecture of problems and portable parallel software systems. Technical Report SCCS-134, Syracuse Center for Computational Science, 1991.

[Hig93] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Draft, May 1993.

[HKT91] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Supercomputing '91*, New Mexico, November 1991.

[Jef85] D.R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3), July 1985.

[KB94] Scott Kohn and Scott Baden. A robut parallel programming model for dynamic non-uniform scientific computations. In *Proceedings of the Scalable High Performance Computing Conference*, Knoxville, TN, May 1994.

[KC81] J. Misra K.M. Chandy. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(11), April 1981.

[LKMS91] S. Lin, E. Kuh, and M. Marek-Sadowska. SWEC: A stepwise equivalent conductance simulator for cmos vlsi circuits. In *Proc. of European Design Automation conference*, February 1991.

[Lun94] Steve Luna. Implementing an efficient portable global memory layer on distributed memory multiprocessors. Master's thesis, Computer Science Division, University of California at Berkeley, 1994.

[SBW91] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and run-time compilation. *Concurrenty: Practice and Experience*, December 1991.

[SK91] Wei Shu and L.V. Kalé. Chare kernel – a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11:198–211, 1991.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *International Symposium on Computer Architecture*, 1992.

[WS93] M.S. Warren and J.K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Supercomputing '93*, pages 12–21, Portland, Oregon, November 1993.

[WY93] Chih-Po Wen and Katherine Yelick. Parallel timing simulation on a distributed memory multiprocessor. In *International Conference on CAD*, Santa Clara, CA, November 1993. An earlier version appeared as UCB Technical Report CSD-93-723.

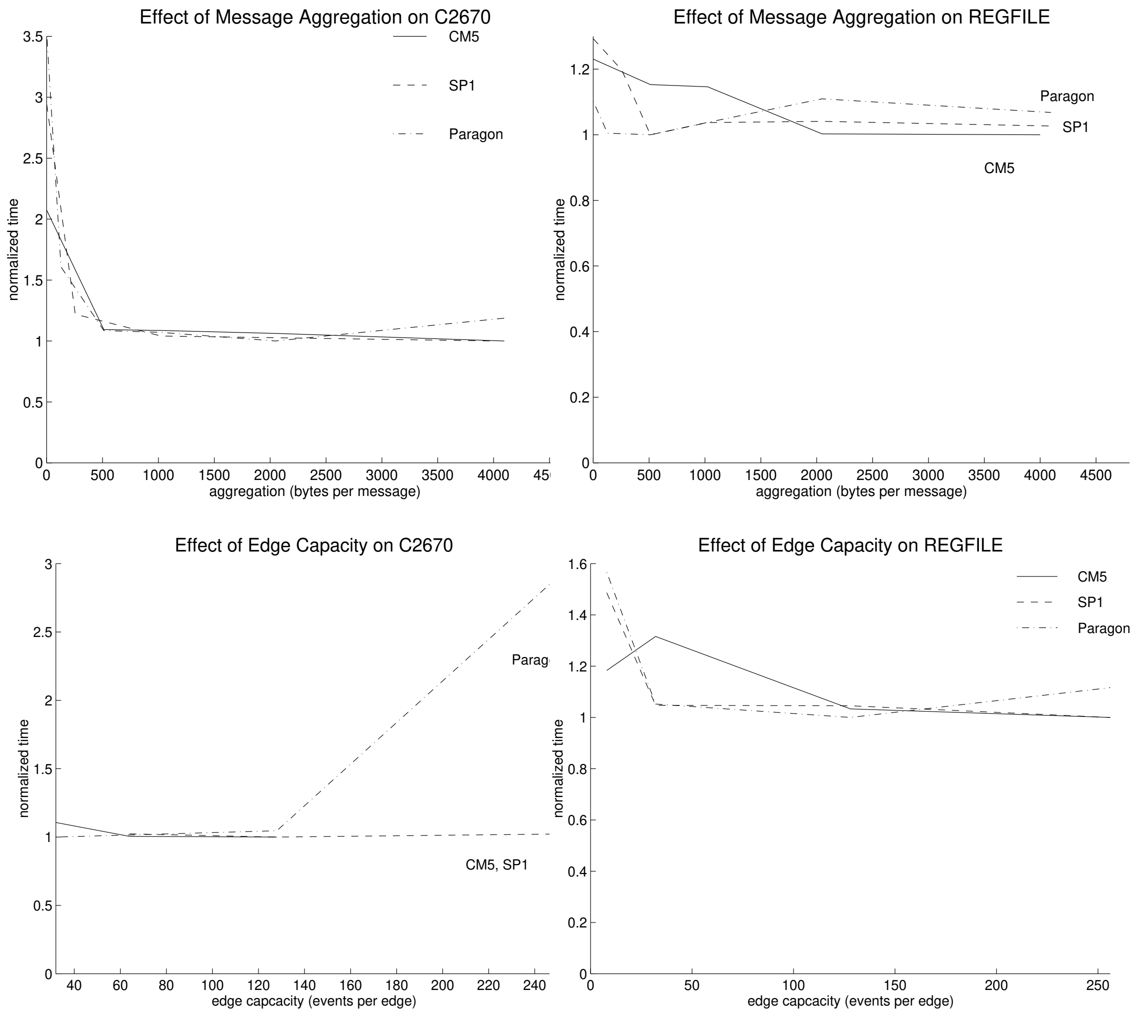

Figure 6: Impact of message aggregation and edge capacity on performance.