BOS is Boss: A Case for Bulk-Synchronous Object Systems

Mark W. Goudreau* Kevin Lang† Girija Narlikar‡ Satish B. Rao§

Abstract

A key issue for parallel systems is the development of useful programming abstractions that can coexist with good performance. We describe a communication library that supports an object-based abstraction with a bulk-synchronous communication style; this is the first time such a library has been proposed and implemented. By restricting the library to the exclusive use of barrier synchronization, we are able to design a simple and easy-to-use object system. By exploiting established techniques based on the bulk-synchronous parallel (BSP) model, we are able to design algorithms and library implementations that work well across platforms.

1 Introduction

Portable parallel programming systems should provide useful abstractions without precluding efficient execution. This paper describes a step towards this goal through the use of a communication library called the BSP Object System (BOS). BOS provides the convenience of efficient shared objects in a system optimized for (and restricted to) batch communication. To our knowledge, this is the first communication library to provide such mechanisms. The library is easy to use and effective for all the applications that we studied.

It is generally agreed that a shared-address space is a useful abstraction. Many systems for distributed-memory environments have focused on providing a shared-address space with communication at the page level [13, 33, 34, 35] or at the user-defined object level [4, 5, 8, 16, 41, 47, 49]. Another useful abstraction is to define simple communication and synchronization mechanisms. A technique we call *batch communication*, based on the BSP model [43], restricts access to communicated data until after a barrier synchronization. For the remainder of this paper, we use the term *synchronous* to describe systems that utilize batch communication exclusively, and *asynchronous* to describe all other systems. Batch communication has influenced the design of several communication libraries, including Oxford BSP [37], Green BSP [28], and BSPlib [31].

We claim that both application programming and system implementation are simpler using BOS than using other techniques that provide a shared-object space. Under most shared-object systems, processes refer to consistent objects through the use of a variety of synchronization mechanisms (e.g. locks, semaphores, and barriers). The flexibility and complexity of such systems make them difficult to program.

Both theoretical and experimental projects have demonstrated that synchronous systems are sufficient in terms of performance [6, 25, 26, 27, 28, 29, 30, 36, 43, 44]. Furthermore, an explicit and theoretically justifiable cost model can be utilized for synchronous systems, which can provide clear guidance to the application programmer and the system implementer. Such cost models can be used to predict (or bound) execution times, memory utilization, or communication requirements for parallel programs running on real platforms. Other programming approaches that emphasize the use of a cost model include the NESL [9] and Cilk [10, 11] languages, which utilize a work-depth cost model, and the Split-C language [20], which utilizes the LogP cost model [21].

In comparison to other approaches that provide shared-memory communication in a synchronous environment, BOS does not require preallocation for each shared data structure by each process. This can greatly reduce memory usage, as many objects need only be accessed by a subset of the processes. In comparison to approaches that provide message-passing communication in a synchronous environment, BOS provides the convenience of a shared-memory abstraction and facilitates the reuse of code.

We support these claims concerning the ease and efficiency of BOS by writing several application programs: electromagnetic particle simulation, Cholesky factorization, shortest path computation, circuit simulation, and Groebner basis calculation. We describe performance characteristics of these applications on two parallel platforms: an SGI Challenge and a network of PCs. Since BOS restricts the synchronization mechanism, library implementation is greatly simplified. For example, the entire BOS system was imple-

*C&C Research Laboratories, NEC USA, Inc., 4 Independence Way, Princeton, NJ 08540. goudreau@ccrl.nj.nec.com

†NEC Research Institute, 4 Independence Way, Princeton, NJ 08540. kevin@research.nj.nec.com

‡School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213. girija@cs.cmu.edu

§NEC Research Institute, 4 Independence Way, Princeton, NJ 08540. satish@research.nj.nec.com

mented in 1600 lines of code.[1]

The remainder of the paper is organized as follows. Related work is in Section 2. The BSP model and BOS are described in Sections 3 and 4, respectively. Applications are described in Section 6. Experimental results are analyzed in Section 7, and conclusions are in Section 8.

2 Related Work

One approach for providing a shared-memory abstraction for a distributed system is shared virtual memory (SVM), which provide communication at the level of pages [35]. Though such systems can be efficient for some applications, performance is highly sensitive to the interaction of data structures and page sizes [33]. By sharing data at the page level, extensive false sharing can occur.

This inefficiency can be substantially reduced by providing communication between processes at the level of user-defined objects [4, 5, 8, 16, 47]. To our knowledge, none of these systems rely on the exclusive use of barrier synchronization. Of particular relevance to our work is Scales and Lam's SAM system [41], which provides a shared-memory abstraction in conjunction with primitives that allow many of the performance advantages of a message-passing system. SAM ties synchronization to data accesses, allows prefetching of data and pushing data to remote processes, and provides chaotic access.[2] Object communication is explicitly visible to the programmer; the programmer may therefore attempt to optimize the program by reducing communication costs.

Yelick et al.'s Multipol (Multi-ported object library) project [49] provides a variety of data structures for irregular applications running on distributed-memory machines. Examples include trees, graphs, and sets. The programmer assumes an event-driven model, where each process repeats a scheduling loop that looks for available work via interactions with the underlying data structures. Communication and partitioning issues are handled by the data structure implementer, and can to a large extent be hidden from the higher-level programmer.

Culler et al.'s Split-C programming language [20] also supports shared objects through the use of global pointers. Although Split-C is asynchronous, many efficient application programs written with the language rely on batch communication [20, 23].

Synchronous communication libraries have to date assumed a single program, multiple data (SPMD) programming approach with primitives to support either message passing or direct access to remote memory locations. Hill et al.'s BSPlib library support both of these approaches [31]. The BSPlib remote memory access, however, requires each process to preallocate memory for each shared object, whether that object is used by the process or not. Further, the approach relies on an object registration technique whereby all processes must agree on the size of each object *a priori*. The BSPlib message passing is quite efficient, but does not provide a shared-memory abstraction and has limitations with respect to the reuse of data structures.

3 The BSP Model

The BOS design is based on the BSP (Bulk-Synchronous Parallel) model [43]. Under BSP, a parallel machine consists of a set of processors, each with its own local memory, and an interconnection network that can route packets of some fixed size between processors. The computation is divided into *supersteps*. In each superstep, a processor can perform operations on local data, send packets, and receive packets. A packet sent in one superstep is delivered to the destination processor at the beginning of the next superstep. Consecutive supersteps are separated by a barrier synchronization of all processors.

The communication time of an algorithm in the BSP model is given by a simple cost function. The three basic parameters that model a parallel machine are: (i) the number of processors p, (ii) the *gap* g, which reflects network bandwidth on a per-processor basis, and (iii) the *latency* L, which is the minimum duration of a superstep, and which reflects the latency to send a packet through the network as well as the overhead to perform a barrier synchronization.

Consider a BSP program consisting of S supersteps. Then the execution time for superstep i is given as:

$$w_i + gh_i + L \qquad (1)$$

where w_i is the largest amount of work (local computation) performed, and h_i the largest number of packets sent or received, by any processor during the ith superstep. The execution time of the entire program is:

$$W + gH + LS \qquad (2)$$

where $W = \sum_{i=0}^{S-1} w_i$ and $H = \sum_{i=0}^{S-1} h_i$. We call w_i and W the *work depths* of the superstep and the program, respectively.

Thus, efficient programming of a BSP machine is based on several simple principles. To minimize the execution time, the programmer must (i) minimize the work depth of the program, (ii) minimize the maximum number of packets sent or received by any processor in each superstep, and (iii) minimize the total number of supersteps in the program. In practice, these objectives can conflict, and trade-offs must be made. The correct trade-offs can be selected by taking into account the particular g and L parameters of the underlying machine.

4 The BSP Object System (BOS)

In the BOS system, all shared objects are addressed using global object ids. Each process maintains a hash table of shared objects, which we call the *object cache*; the hash table is indexed by applying a hash function to object ids. Each shared object has an *owner* process, in whose memory space it is initially created. Other processes may store copies of the object in their object caches, and are therefore called *readers*. As the computation proceeds, each process must know during the current superstep i, which objects it needs in the next superstep ($i+1$). It can then issue requests for a copy of each of these objects during superstep i. At the beginning of superstep $i+1$, a local and consistent copy will be made available to the requesting process. The process can then access the copy during superstep $i+1$ using the corresponding object-id. As with the BSP model, we assume a SPMD programming model with one process fixed on each processor.

The global namespace for the objects simplifies the job of the programmer. In theory, the programmer need not distinguish between local and remote objects. However, in practice, the programmer will in general still need to reason about which objects are local and which are non-local, since communication incurs a cost under the BSP model. Nevertheless, the programmer does not need to know where (in which process's cache) the remote objects reside, and may simply request an updated copy of a remote object using its object id.

The functions that constitute the BOS interface are described below.

- `OBJ_begin`: Starts program with the number of processes requested.

[1] This includes comments, but not the code for the underlying BSP library.

[2] Chaotic algorithms do not need to use the most current data. More relaxed communication and synchronization therefore becomes possible.

- `OBJ_end`: Called by all processes at the end of the program.
- `OBJ_generate_ids`: Each object in the shared-memory space requires a unique object id. The user can either choose an id, or call this function to obtain a block of previously unused ids.
- `OBJ_create`: Used to create an object with a certain object id and number of bytes. Function call returns a pointer to system space where the object will be maintained. The process can then proceed to initialize the value of the object.
- `OBJ_cache_new`: Requests a fresh copy of an object with the requested id. The BOS library keeps track of the object's owner, and sends it a request for a fresh copy. If it does not yet know who the owner is, it sends out a broadcast request that is answered by the owner. A coherent copy will be available in the calling process's object cache at the beginning of the next superstep.
- `OBJ_get`: Returns a pointer to the object requested. It is typically used after `OBJ_cache_new` to request an object in superstep i, and `OBJ_get` is used to access the object in superstep $i+1$. The data in the object can be directly accessed through the pointer returned.
- `OBJ_owner_update`: The owner of an object (i.e., the process that creates it) can update *all* remote copies of an object by using this function. The object library keeps track of the current set of readers and forwards the updates to them. Updates are visible at the beginning of the next superstep.
- `OBJ_free`: Removes the object with the specified object id from the calling process's object cache. If the calling process was a reader, the object library simply informs the owner of the object that the process is no longer a reader. If the process was the owner, the object library informs all the readers of the object that the object does not exist and should be removed from their caches. The object does not exist in the system after the current superstep.
- `OBJ_sync`: The barrier synchronization function call. After the call, all outstanding requests must be satisfied.
- `OBJ_nprocs`: Returns the number of processes.
- `OBJ_myid`: Returns the process id, in the range of 0 to one less than the number of processes.
- `OBJ_rpc`: Requests remote execution of a specified procedure on an object with the specified id. An additional, variable length argument may be sent. The procedure is executed locally by the owner of the object before the start of the next superstep. If there are several `OBJ_rpc` calls on the same object all the procedures are executed in an arbitrary but serialized order.
- `OBJ_modify_remote`: Allows a reader of an object to request the execution of a procedure that modifies the value of an object in its owner's object cache based on the reader's local copy. If several requests are made by one reader in the same superstep, only the last request is sent by the object library at the end of the superstep. This allows the reader to modify an object's local copy several times, while only one message is sent at the end of the superstep to modify the owner's original copy. As with `OBJ_rpc`, the procedure is executed before the start of the next superstep. If more than one reader calls `OBJ_modify_remote` on the same object, they are all executed in an arbitrary but serialized order.

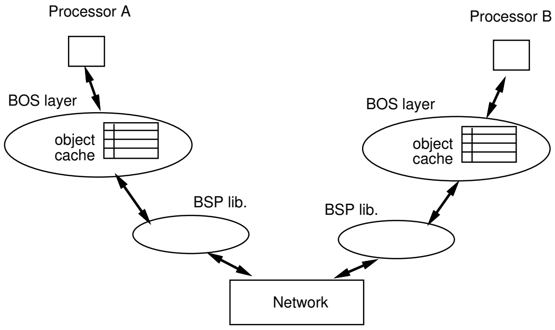

Figure 1: The BOS object system. The BOS layer is built on top of a regular BSP library, which is used for communication between processors. The BOS layer maintains the object cache, which can be accessed only through BOS function calls.

The BOS library is built on top of a regular BSP library, and is therefore easily portable to any platform that supports a BSP library. The object layer implements the above BOS functions by maintaining the object cache at each processor, and using the underlying BSP library to send messages when required (see Figure 1). Each entry in the object cache, which is not directly accessible to the programmer, contains information about a BOS object. The information includes the object id, the state and owner of the object, a pointer to a local copy in the processor's memory, and a bit vector indicating the current readers of the object (used only by the owner). Each BOS-level barrier synchronization (`OBJ_sync`) is implemented using two barrier synchronization calls (`bsp_sync`) of the underlying BSP library, that is, each BOS superstep corresponds to two supersteps of the BSP library. This allows the BOS layer to make requests for objects or remote function invocations and receive the response before the next BOS superstep. For example, when a processor requests an updated copy of a previously cached object (using `OBJ_cache_new`), the BOS layer looks up the owner of the object in the object cache, sends the appropriate message to the owner, and calls `bsp_sync`. The owner then replies, and the requesting processor receives the reply and updates its local copy after another `bsp_sync`. Figure 2 shows a simple code fragment to contrast the interface of the BOS system with a typical asynchronous object system.

5 Platforms

We used two platforms for our experiments. The first platform was a 16-processor SGI Challenge. This machine has relatively slow processors (150 MHz R4400s) but sophisticated bus-based shared memory hardware. Our BSP library used shared memory to communicate. The second platform was a cluster of 16 200-MHz x686 PCs running Linux and connected by Myrinet network switches; a more complete description of this platform can be found in [1, 24]. On the PC cluster our BSP library used VMMC [22] to communicate. BSP g and L parameters for both platforms are shown in Table 1. We note that the two platforms have quite comparable gap parameters, while the SGI has a somewhat superior latency parameter.

6 Applications

In this section, we discuss five applications that use the BOS communication primitives. For each application, we attempt to explain how barrier synchronization drives the algorithm design, and also

UPDATING SHARED OBJECTS

(a) In an asynchronous system, shared objects are updated by obtaining locks. Each request to a remote object incurs communication, and the process typically blocks until the communication is completed.

```
obtain(lock A)     // incurs communication
  Obj A = ...;
release(lock A)

obtain(lock B)     // incurs communication
  Obj B = ...;
release(lock B)

obtain(lock C)     // incurs communication
  Obj C = ...;
release(lock C)
```

(b) With a synchronous system, all the local copies are first updated, and then requests are made to propagate the updates to the original copy. The original copy of object will be updated after the barrier synchronization. Conflict is avoided by writing the code so that no other processes attempt to update the objects during the same superstep.

```
Obj A = ...;
OBJ_modify_remote(Obj A);

Obj B = ...;
OBJ_modify_remote(Obj B);

Obj C = ...;
OBJ_modify_remote(Obj C);

OBJ_sync;    // incurs communication
```

Figure 2: A coding example.

	PC, VMMC		SGI, shared memory	
nprocs	bandwidth (μsec/byte)	latency (μsec/step)	bandwidth (μsec/byte)	latency (μsec/step)
2	0.087	56	0.139	24
4	0.101	182	0.141	29
6	0.111	295	0.148	35
8	0.121	414	0.142	41
10	0.126	532	0.145	47
12	0.141	650	0.147	52
14	0.168	781	0.149	104

Table 1: BSP parameters for our platforms.

how objects are used. Note that optimization based on the BSP cost model, including load balancing, communication balancing, and synchronization reduction are done by the application programmer.

6.1 Electro-Magnetic Particle-In-Cell Simulation

The electro-magnetic particle-in-cell (EMPIC) application simulates the movement of charged particles that exert electric and magnetic forces on each other [18, 32, 46]. We use a standard particle-grid method which discretizes the space on a grid, and uses a leapfrog integration scheme to solve Maxwell's differential equations across discrete timesteps. Each timestep consists of four phases. In the *scatter* phase, the current-density of each grid point is computed from the particles in the grid cells around it. In the *solve* phase, new values of electric and magnetic fields are computed at each grid point, based on the old field values and the current density computed in the scatter phase. In the *gather* phase, the force on each particle is computed from the field values at the grid points around it, and in the *push* phase this force is used to update the particle's position and velocity. A particle may move between grid cells as a result of the push phase.

We started with the sequential C code written by I. Ashok, which was adapted from D. Walker's sequential Fortran code. In our BOS version, we define the electric and magnetic fields and the current density at each grid point as separate BOS objects. Each grid point is associated with the list of particles located in one of its neighboring grid cells; this entire list is also defined as a BOS object, and is local to the processor that owns the grid point. Each object associated with a grid point was assigned an object id derived from the (x, y, z) coordinates of the grid point, so that all the objects associated with a given grid point could be easily accessed.

Parallelizing the original serial code using BOS was straightforward, because each processor simply needs to know the set of grid points in its partition, and the set of grid points that are neighbors of its own grid points but not local to (not owned by) the processor. We implemented two partitioning schemes: in one, the grid is partitioned across processors as slices along the Y-axis, and in the other, it is partitioned into 3D blocks. We first implemented the former scheme; it then took one of the authors another three hours to add the latter scheme (including debugging time). The two schemes simply differ in the functions that step over and cache neighboring, non-local grid points. Consequently, adding a new partitioning scheme and switching between schemes at runtime is fairly simple. Our current implementation does not include dynamic load balancing.[3] However, because each processor keeps track of the boundaries of its own partition, we expect that adding the load balancing will simply involve adjusting the partition boundaries between processors according to the particle distribution. The original sequential program was 1150 lines long; our parallel version with both partitioning schemes has 2050 lines, of which, 200 lines are devoted to caching neighboring grid points according to the partitioning scheme.

Since this simulation proceeds in timesteps, it is well-suited to the BSP style of programming. The advantage of using the BOS library over a regular BSP library is the transparent access to neighboring, non-local grid points. For efficiency, we chose to distinguish between local and non-local objects, that is, a processor always stores the upper and lower bounds of the coordinates of the grid points local to it. However, a processor need not know where the non-local objects it accesses reside. This would be particularly useful when the grid points and particles are moved between processors to balance the load.

[3] This would require a simple, new function to change ownership of an object to be added to the BOS library.

To further explain how the object library was useful, we describe in brief the implementation of some of the phases. In the scatter phase, a processor steps through particles within its partition, and updates the current densities at the surrounding grid points accordingly. However, particles along the edges may affect the current densities of non-local grid points. When the processor finds a current density object to be non-local, it simply updates a local copy of it; several local particles may modify this local copy. At the end of the phase, it uses `OBJ_rpc` to update the original copy of each non-local current density in the memory of its owner processor, and then calls `OBJ_sync` to ensure that the values are updated before the next phase. In the solve phase, a processor first calls `OBJ_cache_new` to cache latest copies of field values of its neighboring, non-local grid points. It then calls `OBJ_sync` to ensure that the objects have been cached, and proceeds to compute local field values based on the neighboring values.

Because the application is highly bulk-synchronous, an implementation using an asynchronous object library (such as SAM) would be very similar, with barrier-like synchronization to separate the phases and ensure that all the data has been communicated. However, since the underlying BSP layer in our object library is designed to combine messages and optimize performance for bulk-synchronous computations, we expect our object library to be better suited for writing such applications. Adhara [3] is a runtime system designed specifically for space-based applications such as EMPIC. Adhara automatically partitions the grid and the particles, and balances the load dynamically. Therefore, coding EMPIC in Adhara is likely to be simpler than using our BOS library. However, Adhara is optimized specifically for space-based simulations, while our object library is designed to be a more general library. We conjecture that the optimizations performed within the Adhara runtime system could be implemented more easily within the application using our BOS layer, compared to a regular BSP library or an asynchronous object layer.

6.2 Dense Cholesky Factorization

A matrix, A, is symmetric positive definite if $A = A^T$ and $x^T Ax > 0$ for every nonzero vector x. The Cholesky factor, L, of a real symmetric positive definite matrix, A, is defined as the lower-triangular matrix that satisfies $A = LL^T$. It is guaranteed that if A is a real symmetric positive definite matrix, then it has a unique Cholesky factorization.

Cholesky factorization is a computation-intensive problem that has inspired several parallel-computing approaches [7, 38, 39, 40]. Efficient factorization of sparse matrices is considerably more complex than for dense matrices. (Though we have begun investigating sparse-matrix factorization, results are not available by the time of the writing of this report.) In this section, we describe a BOS implementation for dense Cholesky factorization, which is relatively straightforward computationally, but takes advantage of the object-level communication.

The algorithm used here is based on a sequential factorization algorithm that transforms the lower-triangular portion of A into L one column at a time, starting from the leftmost column and working right. Calculating element $a_{i,j}$ of L requires knowledge of all the elements of row i up to column $j-1$, as well as all the elements of row j up to column j. Such data dependencies map quite easily into a BOS computation where the (lower-triangular) rows of A are objects that are transferred between processes when necessary. Consider an implementation where one column of L is calculated per superstep, and the rows are mapped to processes in a round-robin manner. When calculating the elements in column j that map to some process x, all that is needed is row j and the rows that are already mapped to x.

From a BSP perspective, this simple approach performs a fairly large number of barrier synchronizations. Furthermore, the communication is not balanced, as each superstep one and only one process is broadcasting a row to all the others. Both problems can be addressed by solving m columns per superstep (where m is typically far less than the number of columns). These advantages are achieved at the cost of redundant computation that allow each process to compute the upper $m \times m/2$ elements of the m columns.

6.3 Multiple Source Shortest Path Algorithms

Given a graph $G = (V, E)$, with weights $w : E \rightarrow R$ and a node $s \in V$, we compute the shortest distances from s to every node in V. We developed code for solving this problem on the same graph simultaneously for a number of sources.

Similar to previous work [28], we use a variant of Dijkstra's algorithm with a heap. This algorithm essentially updates distances locally for a while and then communicates and repeats until all distances are correct. The communication is initiated when each processor either has "locally correct distances" or has done more than a specified amount of work: the "work factor" parameter. This work factor parameter trades off synchronizations (and corresponding items such as data aggregation) against load imbalance and extra computation caused by computing with old values.

We first implemented a single source algorithm as follows. Each node in the graph was an object in the BOS system. Thus, all accesses to nodes of the graph used the object system even in the local portion. This approach made coding of the algorithm very simple, from inputting the graph to organizing the communication. However, because the basic computation unit is small for each node access, it was also very inefficient. For example, it was a factor of two slower than a previous parallel implementation [28] and a factor of five slower than the very efficient sequential implementation of Cherkassky, Goldberg, and Radzhik's SPLIB [17]. The graph representation in SPLIB is tuned carefully to suit the shortest path algorithm.

We therefore adopted a "crush and compute" methodology; that is, we pack local pieces of the graph into SPLIB's representation and use SPLIB's code as our local compute engine. We use our object representation to handle the communication both between processors and to the compute engine. Our single processor version achieves the same running time as the analogous implementation in SPLIB.

We also added the ability to solve the shortest paths problem for many sources simultaneously. (This was also done in [28].) We refer to the code as mCrushSp. This is perhaps a more interesting problem for parallel computing since computing shortest paths for a single source is extremely fast sequentially. Moreover, applications such as circuit retiming and graph partitioning have multiple source shortest path as their inner loops.

Our input data is a set of geometric graphs as in [28].[4] We use a simple heuristic to partition the graph that attempts both to balance the number of nodes on each process and to minimize the number of edges between processes. Finally, for our experiments on mCrushSp, we always ran 50 simultaneous shortest paths.

6.4 Parallel Circuit Simulation

We implemented the parallel conservative discrete event method for numerically simulating a circuit. It is based on the parallel simulator CSWEC [48]. The approach is to decompose the circuit into subcircuits that occasionally communicate. The communication is

[4] Ideally, one would like to study a wider class of graphs but it is perhaps beyond the scope of this paper.

organized via a discrete event graph where each node corresponds to a subcircuit. The event graph delivers timestamped messages between two nodes of the graph in order and schedules nodes that are ready to be simulated.

Wen and Yelick [48] implement such an event graph using the Multipol system. Since their system is asynchronous (indeed, parallel discrete event simulation in general is viewed as asynchronous) the event graph's semantics are complex. Moreover, they provided no way to reason about its performance.

The interface for an event graph with bulk-synchronous semantics is much simpler. Moreover, the bulk-synchronous model allows one to reason about its performance. As for the shortest path, we use a work factor parameter to set the tradeoffs between synchronization and processor utilization.

We used an 8-bit ripple carry adder as the basis of our input (it was included with the Multipol distribution.) We made copies of the circuit tied to the same power and ground to generate larger circuits. Note that the circuit pieces are distributed randomly, so we definitely do not simulate the separate copies on distinct processors.

6.5 Groebner Bases

Buchberger's Ph.D. dissertation [12] describes an algorithm for the calculation of a Groebner basis from an input basis of polynomials. A Groebner basis generates the same ideal as the input basis, but has certain properties that make its use preferable for numerous applications, including the ideal membership problem and the solving of a system of algebraic equations. Interestingly, Buchberger's algorithm is quite simple, yet analyzing its computational complexity is difficult. The speed at which a solution is reached is highly dependent on the ordering of operations. The best ordering is not obvious, though certain heuristics have proven to be useful in practice. A full description of Groebner bases and the Buchberger algorithm is beyond the scope of this paper; for more details, we refer the interested reader to Cox, Little, and O'Shea [19].

There has been a fair amount of work on parallelization of Buchberger's algorithm. Vidal describes an algorithm designed and tested on a shared-memory multiprocessor [45]. Chakrabarti and Yelick describe an algorithm suitable for a distributed memory multiprocessor [14] and prove its correctness in [15]. Sodan et al. [42] describe experimental results for a multithreaded architecture implemented on a distributed-memory machine.

In this section, we describe a very simple parallel version of Buchberger's algorithm that is suitable for a synchronous system. The basic algorithm is as follows. It consists of a number of *phases*. In each phase, queued pairs are checked in parallel, and at most one s-polynomial is added to the basis.

1. Begin with $F = (f_1, \ldots, f_p)$, an ordered set of p input polynomials. This set of polynomials is read in by process 0, who creates an object for each of them. All other processes cache copies of F.

2. Each process examines a distinct subset of the $p(p-1)/2$ pairs of input polynomials. These pairs are put into the process's local queue.

3. Each process removes one pair from its local queue and calculates the s-polynomial. It then calculates the normal form of the s-polynomial with respect to the ordered set F. For process i, call the normal form r_i.

4. Each process now informs process 0, which is acting as a centralized controller, whether its r_i is non-zero.

5. If one or more of the normal forms is non-zero, process 0 selects one or more to add to F. All processes are informed of the decision, and receive a copy of the selected polynomial. The new pairs that are formed as a result of the addition are distributed among the processes, and added to their queues.

6. If any process's queue is not empty, go to step 2. Otherwise, terminate. The resulting set of polynomials is a Groebner basis.

The algorithm assumes that each process maintains an entire up-to-date copy of the basis, F. Since the polynomials are large (typically several hundreds to thousands of bytes), this is expensive. However, the basis set is so fundamental to the algorithm, there is a strong intuitive case for replicating the basis. Chakrabarti and Yelick considered the issue in [14], and report that partitioning the basis hampers load-balancing, reduces available parallelism, and suffers from high communication overhead. They conclude that replicating the basis is superior if permitted by memory capacity.

In contrast, the queue is partitioned among the processors. Each process has its own queue of distinct s-polynomials to normalize relative to the basis. To save memory space, the queue actually does not store the s-polynomials. Rather, the a queue entry holds pointers to its two source polynomials from the basis; the s-polynomial is only calculated when the element is dequeued.

The basic prioritization used is very close to the one proposed by Buchberger; it orders pairs based on the degree of the leading term of the s-polynomial. As mentioned earlier, the speed of the algorithm is heavily dependent on the order in which pairs are checked. To make the parallel versions perform ordering as close as possible to the sequential version, we further prioritized the local queues such that if more than one polynomial pair has the same priority, the one that was first added to the queue would be checked first. The experiments described in Section 7 allow each process to try to reduce only one polynomial per step.

Note that the polynomials themselves are rather complex objects, and we therefore achieve some convenience from the use of the object library. However, once added to the basis, the polynomial will not change during the program (unless some post-processing is done to minimize the Groebner basis).

Our implementation makes heavy use of Groebner basis libraries that had been previously developed. In particular, we use code available through UC Berkeley's Multipol project, which in turn utilized serial libraries for polynomial arithmetic developed at CMU.

It is instructive to compare this synchronous Groebner basis algorithm with the asynchronous approaches described by Chakrabarti and Yelick [14] and Sodan et al. [42]. Of all the applications we have examined to date, this one would seem to have the most to gain from an asynchronous implementation. Since the communication required is relatively small, the main performance issue becomes load balancing. For applications where the basic computational task have predictable execution times, or where we can utilize slackness to help with load balancing, synchronous systems are quite suitable. But for Groebner basis calculation, the execution time of the basic computational task (reducing a polynomial relative to a set) is unpredictable, and performing multiple reductions per superstep can cause problems since the addition of a polynomial to a basis set can have an immediate effect on all subsequent tasks. On the other hand, the asynchronous approaches have the disadvantage of being inherently nondeterministic; the execution time depends heavily on the ordering of the tasks, as well as the order in which processes become aware of additions to the basis set.

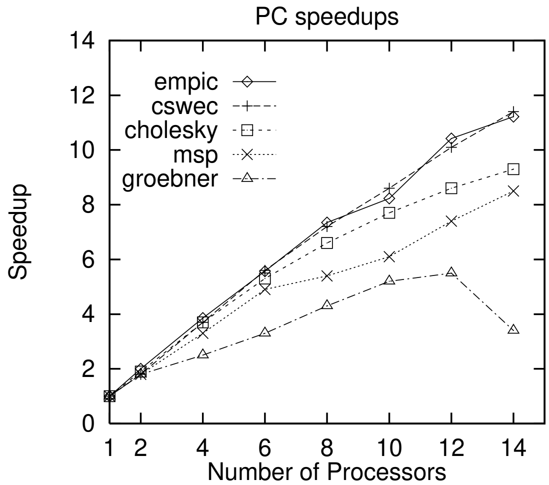

Figure 3: Sample speedups for the five applications on the network of PCs. Input datasets that resulted in the longest execution times were used to generate these numbers.

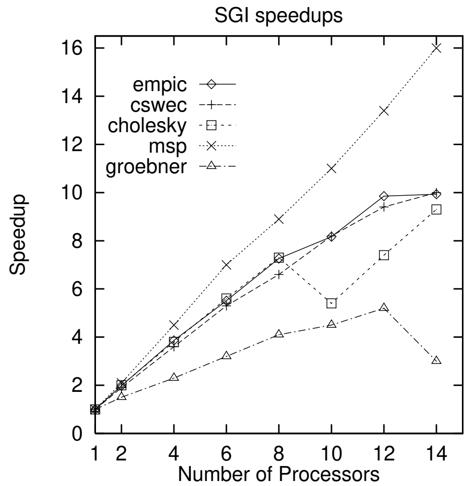

Figure 4: Sample speedups for the five applications on the SGI Challenge. Input datasets that resulted in the longest execution times were used to generate these numbers.

7 Results and Discussion

Figures 3 and 4 show the speedups for each application with sample inputs on the two execution platforms. Speedups shown are with respect to the single processor execution time of the parallel program. For our applications, this is a reasonable sequential program in terms of performance. Detailed results for each application are presented and discussed below.

Results for the EMPIC program are summarized in table 2. Note that we get good speedups on both the network of PCs and the SGI for larger inputs and 14 processors. These results are comparable to the results of Ashok [2], although we report results for a fewer number of timesteps (20). The particles have a higher initial velocity in the X direction; therefore, we partitioned the grid across processors into slices in the Y direction. With this partitioning, we did not see any significant load imbalance across processors. In contrast, block partitioning, that is, partitioning across all three dimensions, did not perform as well due to load imbalance in the X direction (we do not report the results here). Future work involves either adding dynamic load balancing (e.g., by adding a change owner function to the library), or overpartitioning of the grid.

The Cholesky results are summarized in table 3. This application illustrates the possible performance tradeoffs based on the g and L parameters of the platform. The PCs have larger L, and therefore have more to gain by reducing the number of supersteps, in this case by increasing the number of columns processed during each superstep (i.e. m). For the smaller problem size on 14 processors, the program achieves speedups of 4.8 (with $m = 4$) on the PCs and 6.0 (with $m = 1$) on the SGI. For the larger problem size on 14 processors, the program achieves speedups of 9.3 (with $m = 2$) on the PCs and 9.3 (with $m = 1$) on the SGI. Also notice that the PCs raw performance is a factor of 10 better than the SGI's.

Our multiple shortest path results are given in table 4. We get excellent performance even for the small problem sizes. Indeed, we get superlinear speedup on the large problem sizes on the SGI; for example, the speedup is 16 on 14 processors for u100k. This is consistent with results reported in [28] on shortest path computations. Our results, however, are better than those in [28], since our single-processor BOS program is competitive with sequential implementations in SPLIB [17].

Our results for the circuit simulator are summarized in tables 5 and 6. Our sequential performance is essentially the same as that of CSWEC. We were unable to compare our results directly to theirs since we have, as yet, been unable to obtain their test circuits. (Input formats for available circuits are widely varying.) In order to evaluate the parallel performance of CSWEC, the CSWEC program distributed by Wen and Yelick measured what they called the "concurrency": this is the average number of items waiting in the queue upon each item's removal in a single-processor execution. This is a rough upper bound on the maximum available parallelism. CSWEC reported concurrency numbers of 25 for ripple, 38 for twiceRipple, and 214 for ripple16. For twiceRipple we were able to get a speedup of 6.6 on a 10-processor PC farm; thus, we get reasonable speedups even with a parallel slackness of 4.

The results of the Groebner basis experiments are described in Table 7. Three benchmark input bases were used. Out of the standard benchmarks we could find, we chose those that required the longest execution time on one processor. Even so, the initial bases of these benchmarks are quite small—none contain more than five polynomials—and the potential for parallelism is somewhat limited. Our results demonstrate consistent speedup, achieving approximately 50% efficiency for larger problem sizes. It is worth noting that, as with the circuit simulator, the amount of available parallelism in the Groebner benchmarks is somewhat limited. For example, Sodan et al. show that only 75 to 168 tasks need to be handled for their three benchmark programs [42] (one of which is the katsura4 benchmark, which we also use). In terms of relative performance, Chakrabarti and Yelick's data is somewhat less extensive [14], making direct comparisons difficult. For eight benchmarks, they show speedups between 4.5 and 32.4 on 10 processors on a CM-5, in three of the eight cases achieving superlinear speedup (12.5, 21.3, and 32.4). Again, this is reasonable considering the dependence on the ordering of queues. We often observed this effect before rewriting the code to make the ordering closer to the sequential ordering. The results described by Sodan et al. are somewhat superior, as they obtain close to linear speedup for up to approximately 10 processors for three benchmarks, with performance degrading for more processors. As mentioned earlier, the asynchronous nature of their approach leads to a wide range of execution times for different tests on the same data; some of their runtimes vary by a factor of up to 7 for the same input on the same number of processors. This is not the case with the BOS implementation, which is deterministic.

To demonstrate the use of the BSP cost model, Table 8 contains communication and synchronization cost estimations for two applications. The two applications were chosen because they have relatively high communication and synchronization (i.e., overhead) costs. For "empic-small," the overhead costs are dominated by communication; since the PCs and SGI have similar g parameters, their overhead costs in terms of absolute time are quite similar. The overhead percentage, however, is larger for the PCs since the PC processors are faster. In contrast, the "msp-w100-sp-u10k" overheads are dominated by synchronization costs, giving the SGI a performance advantage in terms of absolute time of overhead costs.

Finally, we made measurements on the overhead of the object system. First, we calculated the overhead cost of transferring random h-relations with BOS in comparison to BSPlib message-passing mechanisms. The BSPlib message-passing used "sends" to transmit the data (one-way communication), while BOS used "gets" (two-way communication). Note that "get" is the more powerful primitive. If we assume the computational overhead of BOS is small, we expect BSPlib message-passing to be approximately a factor of 2 faster than BOS for small packet sizes. This is because a phase of requests is followed by a phase of data transfers. For large message sizes, we expect the cost of communication to swamp the cost of the request overhead. The experiments were run on the SGI where $p = 14$ processors and the h-relation size was 100 packets. For a packet size of 32 bytes, the BOS running time was a factor of 3 larger than the BSPlib message passing runtime—we call this factor the overhead ratio. For 256 bytes, the overhead ratio was 1.9; for 1024 bytes, the overhead was 1.4. The experimental results match our expectations of the general trend.

We also ran overhead experiments for the EMPIC application. Recall that each grid point is an object for this application. We compared it to a sequential program that does not use the objects to represent grid points. The overhead ranged from 15% on the large input to 25% on the medium input. The overheads should be much smaller for the other applications since they use far fewer objects during the course of computation.

8 Conclusions

This work may be viewed as a further step toward a cohesive approach to parallel system design. We defined and implemented a library for object-based communication. In comparison to other object-level communication libraries, such SAM and Multipol, it seems clear that reasoning based on the bulk-synchronous model is simpler. Our own experiences, and the experiences of colleagues, continually reinforce this conclusion. In addition to ease of programming, we demonstrated good performance for a range of applications on two platforms: a symmetric multiprocessor and a network of PCs.

Our preliminary parameterization of applications based on the BSP cost model indicates that communication and synchronization costs are not the limiting factors in the performance of our applications on these two platform, at least for large problem sizes. This indicates that further refinement of these algorithms should focus on improving load-balancing. As mentioned earlier, the application programmer is responsible for load-balancing with the BOS system, but for some of our applications the granularity of a computation unit is variable, making the problem difficult. For certain applications, it may be advantageous to investigate a strictly periodic barrier synchronization mechanism, rather than one that relies on processors completing a predetermined number of computation units before requesting synchronization.

The machine parameterization also contradicts the popular belief that "barrier synchronization is too expensive." The SGI with 14 processors is able to perform 10,000 barrier synchronizations per second; the network of PCs with 14 processors can handle 1,200 barrier synchronizations per second. Our experience indicates that applications that are written with care are unlikely to incur large penalties due to barrier synchronization.

We also note the promising overall performance of the network of PCs, which provided speedups similar to the SGI. A synchronous programming discipline is an excellent match for such scalable architectures, which are likely to be prominent in the future.

Finally, we note that the system was easy to program, especially with respect to asynchronous systems. For example, consider the rigors of programming the (asynchronous) Multipol system [49]. A Multipol program uses multiple long-lived threads on each processor to hide latency. The threads access distributed data structures through a well-defined set of operations. Generally, threads are suspended if necessary to wait for the results of these data accesses. This suspension of threads, the associated context savings, and the synchronization requirements must be managed explicitly by the application programmer. Furthermore, the scheduling of threads relies on customized schedulers provided by the programmer. In contrast, BOS utilizes batch communication, which allowed us to reason at the level of a batch of coordinated data access rather than individual data access; this simplifies programming considerably.

Acknowledgments

We thank Richard Alpert for helping us utilize the network of PCs.

References

[1] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin, "User-space communication: A quantitative study," in *SC98: High Performance Networking and Computing*, (Orlando, FL), November 1998.

[2] I. Ashok, "Runtime support for dynamic space-based applications on distributed memory multiprocessors," Tech. Rep. 94-12-03, University of Washington, Seattle, WA, Dec. 1994.

[3] I. Ashok and J. Zahorjan, "Adhara: Runtime support for dynamic, space-based," in *Proceedings of the Scalable High Performance Computing Conference*, May 1994.

[4] H. E. Bal, M. F. Kaashoek, and A. S. Tenenbaum, "Orca: A language for parallel programming of distributed systems," *IEEE Transactions on Software Engineering*, vol. 18, no. 3, March 1992.

[5] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon, "The midway distributed shared memory system," in *COMPCON 1993*, March 1993.

[6] R. H. Bisseling and W. F. McColl, "Scientific computing on bulk synchronous parallel architectures," in *Proceedings of the 13th IFIP World Computer Congress* (B. Pehrson and I. Simon, eds.), vol. 1, pp. 509–514, Elsevier, 1994.

[7] R. H. Bisseling, "Sparse matrix computations on bulk synchronous parallel computers," in *Proceedings of the International Conference on Industrial and Applied Mathematics*, (Hamburg), July 1995.

[8] R. D. Bjornson, *Linda on Distributed Memory Multiprocessors*. PhD thesis, Yale University, Department of Computer Science, November 1992.

[9] G. E. Blelloch and J. Greiner, "A provable time and space efficient implementation of NESL," in *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, (Philadelphia, PA), pp. 213–225, 24–26 May 1996.

[10] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "An analysis of dag-consistent distributed shared-memory algorithms," in *8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 297–308, June 1996.

[11] R. D. Blumofe and C. E. Leiserson, "Space-efficient scheduling of multithreaded computations," *SIAM Journal on Computing*, vol. 27, no. 1, pp. 202–229, Feb. 1998.

[12] B. Buchberger, *An Algorithm for Finding a Basis for the Residue Class Ring of a Zero-Dimensional Polynomial Ideal*. PhD thesis, University of Innsbruck, 1965.

[13] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and performance of Munin," in *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pp. 152–164, Oct. 1991.

[14] S. Chakrabarti and K. Yelick, "Implementing an irregular application on a distributed memory multiprocessor," in *Proceedings of the Fourth ACM/SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pp. 169–179, May 1993.

[15] S. Chakrabarti and K. Yelick, "On the correctness of a distributed memory Gröbner basis algorithm," in *International Conference on Rewriting Techniques and Applications*, (Montreal, Canada), June 1993.

[16] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield, "The Amber system: Parallel programming on a network of multiprocessors," in *Proceedings of the Twelfth ACM Symposium on Operating Systems*, pp. 147–158, December 1989.

[17] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest Paths Algorithms: Theory and Experimental Evaluation," *Math. Prog.*, vol. 73, pp. 129–174, 1996.

[18] C.K.Birdsall and A. B. Langdon, *Plasma Physics Via Computer Simulation*. McGraw-Hill, 1985.

[19] D. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms: An Introduction of Computational Algebraic Geometry and Commutative Algebra*. Springer-Verlag, second ed., 1997.

[20] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a realistic model of parallel computation," in *Fourth ACM Symposium on Principles and Practice of Parallel Programming*, pp. 1–12, May 1993.

[21] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken, "LogP: A practical model of parallel computation," *Communications of the ACM*, vol. 39, no. 11, pp. 78–85, November 1996.

[22] C. Dubnicki, A. Bilas, K. Li, and J. F. Philbin, "Design and implementation of virtual memory-mapped communication on Myrinet," in *Proceedings of 11th International Parallel Processing Symposium*, pp. 388–396, April 1997.

[23] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin, "Fast parallel sorting under LogP: Experience with the CM-5," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 8, pp. 791–805, August 1996.

[24] J. Edler, A. Gottlieb, and J. Philbin, "The NECI LAMP: What, why, and how," in *Proceedings of the NEC Research Symposium*, (Berlin), May 1997. To be published.

[25] A. V. Gerbessiotis and C. J. Siniolakis, "Deterministic sorting and randomized mean finding on the BSP model," in *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 223–232, June 1996.

[26] A. V. Gerbessiotis and L. G. Valiant, "Direct bulk-synchronous parallel algorithms," *Journal of Parallel and Distributed Computing*, vol. 22, no. 2, pp. 251–267, August 1994.

[27] P. B. Gibbons, Y. Mattias, and V. Ramachandran, "Can a shared-memory model serve as a bridging model for parallel computation?," in *9th Annual ACM Symposium on Parallel Algorithms and Architectures*, (Newport, Rhode Island), pp. 72–83, June 1997.

[28] M. W. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas, "Towards efficiency and portability: Programming with the BSP model," in *Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 1–12, June 1996.

[29] M. W. Goudreau and S. B. Rao, "Single message vs. batch communication," in *Algorithms for Parallel Processing* (M. T. Heath, A. Ranade, and R. S. Schreiber, eds.), vol. 105 of *IMA Volumes in Mathematics and Its Applications*, pp. 61–74, Springer-Verlag, 1999.

[30] B. Grayson, M. Dahlin, and V. Ramachandran, "Experimental evaluation of QSM, a simple shared-memory model," Tech. Rep. UTCS TR98-21, University of Texas at Austin, November 1998.

[31] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling, "BSPlib: The BSP programming library," *Parallel Computing*, vol. 24, no. 14, pp. 1947–1980, 1998.

[32] R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*. New York: McGraw-Hill, 1981.

[33] L. Iftode, J. P. Singh, and K. Li, "Understanding application performance on shared virtual memory systems," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 122–133, May 1996.

[34] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Tread marks: Distributed shared memory on standard workstations and operating system," in *Proceedings of the Winter 1994 USENIX Conference: January 17–21, 1994, San Francisco, CA*, pp. 115–132, 1994.

[35] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," in *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pp. 229–239, August 1986.

[36] W. F. McColl, "General purpose parallel computing," in *Lectures in Parallel Computation, Proceedings 1991 ALCOM Spring School on Parallel Computation* (A. M. Gibbons and P. Spirakis, eds.), pp. 337–391, Cambridge University Press, 1993.

[37] R. Miller and J. Reed, *The Oxford BSP Library Users' Guide Version 1.0*. Oxford Parallel, 1993.

[38] E. Rothberg and A. Gupta, "An efficient block-oriented approach to parallel sparse cholesky factorization," in *Supercomputing '92 Proceedings*, 1992.

[39] E. Rothberg and A. Gupta, "Efficient sparse matrix factorization on high-performance workstations–exploiting the memory heirarchy," *ACM Transactions on Mathematical Software*, vol. 17, no. 3, pp. 313–334, September 1991.

[40] E. Rothburg and A. Gupta, "Techniques for improving the performance of sparse matrix factorization on multiprocessor workstations," in *Supercomputing '90*, pp. 232–241, 1990.

[41] D. J. Scales and M. S. Lam, "The design and evaluation of a shared object system for distributed memory machines," in *First Symposium on Operating Systems Design and Implementation*, 1994.

[42] A. Sodan, G. R. Gao, O. Maquelin, J.-U. Schultz, and X.-M. Tian, "Experiences with non-numeric applications on multithreaded architectures," in *Sixth ACM Symposium on Principles and Practice of Parallel Programming*, (Las Vegas, NV), pp. 124–135, June 1997.

[43] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[44] L. G. Valiant, "General purpose parallel architectures," in *Handbook of Theoretical Computer Science* (J. van Leeuwen, ed.), vol. A: Algorithms and Complexity, ch. 18, pp. 943–971, Cambridge, MA: MIT Press, 1990.

[45] J.-P. Vidal, "The computation of Gröbner bases on a shared memory multiprocessor," in *Design an Implementation of Symbolic Computation Systems* (A. Miola, ed.), no. 429 in Lecture Notes in Computer Science, pp. 81–90, Berlin: Springer-Verlag, 1990. International Symposium DISCO '90.

[46] D. W. Walker, "Characterizing the parallel performance of a large-scale, particle-in-cell plasma simulation code," *Concurrency, Practice and Experience*, vol. 2, no. 4, pp. 257–288, Dec. 1990.

[47] W. Weihl, E. Brewer, A. Colbrook, C. Dellarocas, W. Hsieh, A. Joseph, C. Waldspurger, and P. Wang, "Prelude: A system for portable parallel software," Tech. Rep. MIT/LCS/TR-519, MIT, October 1991.

[48] C.-P. Wen and K. Yelick, "Portable runtime support for asynchronous simulation," in *International Conference on Parallel Processing*, August 1995.

[49] K. Yelick, S. Chakrabarti, E. Deprit, J. Jones, A. Krishnamurthy, and C.-P. Wen, "Data structures for irregular applications," in *DIMACS Workshop on Parallel Algorithms for Unstructured and Dynamic Problems*, (Piscataway, NJ), June 1993.

problem	nprocs	PC time (sec)	PC speedup	SGI time (sec)	SGI speedup
empic-small	1	20.826	1.00	46.77	1.00
empic-small	2	10.83	1.92	24.15	1.94
empic-small	4	5.88	3.54	13.43	3.48
empic-small	6	4.18	4.98	9.85	4.75
empic-small	8	3.30	6.31	7.71	6.07
empic-small	10	2.90	7.18	6.79	6.89
empic-small	12	2.57	8.10	5.96	7.85
empic-small	14	2.34	8.90	6.11	7.65
empic-medium	1	88.77	1.0	202.22	1.0
empic-medium	2	44.70	1.99	106.11	1.91
empic-medium	4	23.68	3.75	54.11	3.74
empic-medium	6	16.43	5.40	38.15	5.30
empic-medium	8	12.65	7.01	29.70	6.81
empic-medium	10	11.2	7.93	27.27	7.42
empic-medium	12	9.17	9.68	21.48	9.41
empic-medium	14	8.59	10.33	22.54	8.97
empic-large	1	80.33	1.0	184.86	1.0
empic-large	2	40.01	2.00	93.20	1.98
empic-large	4	20.81	3.86	47.90	3.86
empic-large	6	14.42	5.57	33.56	5.51
empic-large	8	10.93	7.35	25.45	7.26
empic-large	10	9.76	8.23	22.63	8.17
empic-large	12	7.71	10.42	18.76	9.85
empic-large	14	7.16	11.22	18.61	9.93

Table 2: Execution times for empic with inputs "small", "medium" and "large." The "small" input uses a $16 \times 32 \times 16$ grid with $32K$ particles. The "medium" and "large" inputs use a $32 \times 32 \times 32$ grid with $128K$ and $256K$ particles, respectively. Input distribution for the particles is uniform. The large instance is run for 10 timesteps while the medium and small instances are run for 20 timesteps.

problem	nprocs	PC time (sec)	PC speedup	SGI time (sec)	SGI speedup
cholesky-200-m-1	1	1.589	1.0	14.796	1.0
cholesky-200-m-1	2	0.834	1.9	7.667	1.9
cholesky-200-m-1	4	0.495	3.2	4.14	3.6
cholesky-200-m-1	6	0.404	3.9	2.913	5.1
cholesky-200-m-1	8	0.388	4.1	3.622	4.1
cholesky-200-m-1	10	0.404	3.9	2.519	5.9
cholesky-200-m-1	12	0.427	3.7	2.754	5.4
cholesky-200-m-1	14	0.463	3.4	2.472	6.0
cholesky-200-m-4	1	1.785	1.0	16.091	1.0
cholesky-200-m-4	2	0.944	1.9	8.884	1.8
cholesky-200-m-4	4	0.58	3.1	5.377	3.0
cholesky-200-m-4	6	0.468	3.8	4.21	3.8
cholesky-200-m-4	8	0.416	4.3	4.765	3.4
cholesky-200-m-4	10	0.395	4.5	3.617	4.4
cholesky-200-m-4	12	0.386	4.6	4.739	3.4
cholesky-200-m-4	14	0.375	4.8	4.096	3.9
cholesky-500-m-1	1	23.82	1.0	223.878	1.0
cholesky-500-m-1	2	12.18	1.9	112.698	2.0
cholesky-500-m-1	4	6.306	3.8	58.454	3.8
cholesky-500-m-1	6	4.461	5.3	39.95	5.6
cholesky-500-m-1	8	3.611	6.6	30.613	7.3
cholesky-500-m-1	10	3.151	7.6	41.603	5.4
cholesky-500-m-1	12	2.904	8.2	30.166	7.4
cholesky-500-m-1	14	2.805	8.5	24.175	9.3
cholesky-500-m-2	1	23.95	1.0	232.451	1.0
cholesky-500-m-2	2	12.27	1.9	118.466	2.0
cholesky-500-m-2	4	6.411	3.7	62.032	3.7
cholesky-500-m-2	6	4.521	5.3	45.152	5.1
cholesky-500-m-2	8	3.61	6.6	45.032	5.2
cholesky-500-m-2	10	3.1	7.7	29.457	7.9
cholesky-500-m-2	12	2.782	8.6	31.006	7.5
cholesky-500-m-2	14	2.571	9.3	33.655	6.9

Table 3: Execution times for dense cholesky factorization for 200×200 matrix and 500×500 matrix. The m parameter specifies the number of columns that are processed in each superstep.

problem	nprocs	PC time (sec)	PC speedup	SGI time (sec)	SGI speedup
msp-w100-sp-u10k	1	3.59	1.0	12.411	1.0
msp-w100-sp-u10k	2	1.96	1.8	6.038	2.0
msp-w100-sp-u10k	4	1.08	3.3	3.269	3.8
msp-w100-sp-u10k	6	0.95	3.8	2.734	4.5
msp-w100-sp-u10k	8	0.87	4.1	2.195	5.7
msp-w100-sp-u10k	10	0.79	4.5	1.884	6.6
msp-w100-sp-u10k	12	0.70	5.1	1.595	7.9
msp-w100-sp-u10k	14	0.71	5.1	1.679	7.4
msp-w200-sp-u10k	1	3.34	1.0	11.032	1.0
msp-w200-sp-u10k	2	1.83	1.8	5.4680	2.0
msp-w200-sp-u10k	4	1.00	3.3	3.1147	3.5
msp-w200-sp-u10k	6	0.88	3.8	2.3729	4.6
msp-w200-sp-u10k	8	0.80	4.2	2.0884	5.3
msp-w200-sp-u10k	10	0.72	4.6	1.818	6.1
msp-w200-sp-u10k	12	0.63	5.3	1.588	6.9
msp-w200-sp-u10k	14	0.66	5.1	1.747	6.3
msp-w200-sp-u50k	1	24.07	1.0	125.937	1.0
msp-w200-sp-u50k	2	12.09	2.0	53.055	2.4
msp-w200-sp-u50k	4	6.43	3.7	24.032	5.2
msp-w200-sp-u50k	6	5.13	4.7	17.233	7.3
msp-w200-sp-u50k	8	4.36	5.5	13.068	9.6
msp-w200-sp-u50k	10	3.82	6.3	11.086	11.4
msp-w200-sp-u50k	12	3.39	7.1	9.520	13.2
msp-w200-sp-u50k	14	3.22	7.5	8.422	15.0
msp-w500-sp-u50k	1	22.04	1.0	115.47	1.0
msp-w500-sp-u50k	2	11.16	2.0	48.633	2.4
msp-w500-sp-u50k	4	5.94	3.7	22.043	5.2
msp-w500-sp-u50k	6	4.69	4.7	15.513	7.4
msp-w500-sp-u50k	8	3.97	5.5	12.710	9.1
msp-w500-sp-u50k	10	3.44	6.4	10.86	10.6
msp-w500-sp-u50k	12	2.98	7.4	8.525	13.5
msp-w500-sp-u50k	14	2.96	7.4	8.413	13.7
msp-w500-sp-u100k	1	48.39	1.0	281.502	1.0
msp-w500-sp-u100k	2	26.21	1.8	132.184	2.1
msp-w500-sp-u100k	4	14.49	3.3	62.202	4.5
msp-w500-sp-u100k	6	9.89	4.9	40.403	7.0
msp-w500-sp-u100k	8	8.90	5.4	31.603	8.9
msp-w500-sp-u100k	10	7.89	6.1	25.542	11.0
msp-w500-sp-u100k	12	6.51	7.4	20.913	13.4
msp-w500-sp-u100k	14	5.69	8.5	17.506	16.0

Table 4: Execution times for multiple shortest path. The work factor follows -w, and the name of the graph follows sp. The graphs u10k, u50K, and u100K have 10 thousand, 50 thousand, and a 100 thousand nodes, respectively.

problem	nprocs	PC time (sec)	PC speedup	SGI time (sec)	SGI speedup
cswec-ripple-200	1	5.34	1.0	15.01	1.0
cswec-ripple-200	2	3.09	1.7	8.26	1.8
cswec-ripple-200	4	1.77	3.0	5.56	2.7
cswec-ripple-200	6	1.37	3.9	3.86	3.9
cswec-ripple-200	8	1.30	4.1	2.98	5.0
cswec-ripple-200	10	1.19	4.5	2.80	5.3
cswec-ripple-200	12	1.15	4.6	3.08	4.9
cswec-ripple-200	14	1.17	4.6	2.47	6.1
cswec-ripple-1000	1	5.27	1.0	15.38	1.0
cswec-ripple-1000	2	2.89	1.8	7.81	2.0
cswec-ripple-1000	4	1.55	3.4	5.33	2.9
cswec-ripple-1000	6	1.31	4.0	4.09	3.8
cswec-ripple-1000	8	1.33	4.0	3.68	4.2
cswec-ripple-1000	10	0.98	5.4	3.63	4.2
cswec-ripple-1000	12	1.18	4.5	3.40	4.5
cswec-ripple-1000	14	1.15	4.6	3.64	4.2
cswec-twiceripple-500	1	10.59	1.0	31.40	1.0
cswec-twiceripple-500	2	5.67	1.9	18.76	1.7
cswec-twiceripple-500	4	3.17	3.3	9.60	3.3
cswec-twiceripple-500	6	2.46	4.3	6.99	4.5
cswec-twiceripple-500	8	1.90	5.6	5.73	5.5
cswec-twiceripple-500	10	1.61	6.6	5.45	5.8
cswec-twiceripple-500	12	1.71	6.2	4.98	6.3
cswec-twiceripple-500	14	1.60	6.6	4.42	7.1
cswec-twiceripple-1000	1	10.59	1.0	31.23	1.0
cswec-twiceripple-1000	2	5.85	1.8	18.85	1.7
cswec-twiceripple-1000	4	3.18	3.3	9.96	3.1
cswec-twiceripple-1000	6	2.33	4.6	7.48	4.2
cswec-twiceripple-1000	8	1.90	5.6	5.87	5.3
cswec-twiceripple-1000	10	1.61	6.6	5.51	5.7
cswec-twiceripple-1000	12	1.75	6.1	5.27	5.9
cswec-twiceripple-1000	14	1.57	6.7	4.94	6.3

Table 5: Execution times for the circuit simulator with inputs "ripple" (with work factors 200 and 1000) and "twiceripple" (with work factors 500 and 1000).

problem	nprocs	PC time (sec)	PC speedup	SGI time (sec)	SGI speedup
cswec-fiveripple-500	1	27.62	1.0	81.14	1.0
cswec-fiveripple-500	2	15.21	1.8	43.89	1.8
cswec-fiveripple-500	4	8.13	3.4	24.01	3.4
cswec-fiveripple-500	6	5.74	4.8	17.10	4.7
cswec-fiveripple-500	8	4.46	6.2	13.84	5.9
cswec-fiveripple-500	10	3.87	7.1	10.56	7.7
cswec-fiveripple-500	12	3.43	8.1	9.66	8.4
cswec-fiveripple-500	14	3.34	8.3	9.02	9.0
cswec-fiveripple-1000	1	27.54	1.0	82.43	1.0
cswec-fiveripple-1000	2	14.83	1.9	42.20	2.0
cswec-fiveripple-1000	4	7.81	3.5	22.85	3.6
cswec-fiveripple-1000	6	5.72	4.8	16.29	5.1
cswec-fiveripple-1000	8	4.20	6.6	12.81	6.4
cswec-fiveripple-1000	10	3.44	8.0	10.96	7.5
cswec-fiveripple-1000	12	3.33	8.3	9.58	8.6
cswec-fiveripple-1000	14	3.32	8.3	9.83	8.4
cswec-ripple16-500	1	97.09	1.0	293.06	1.0
cswec-ripple16-500	2	53.93	1.8	151.92	1.9
cswec-ripple16-500	4	27.56	3.5	80.55	3.6
cswec-ripple16-500	6	19.01	5.1	54.88	5.3
cswec-ripple16-500	8	14.55	6.7	44.19	6.6
cswec-ripple16-500	10	12.44	7.8	35.81	8.2
cswec-ripple16-500	12	10.61	9.2	31.16	9.4
cswec-ripple16-500	14	9.54	10.2	29.27	10.0
cswec-ripple16-2000	1	95.58	1.0		
cswec-ripple16-2000	2	52.52	1.8		
cswec-ripple16-2000	4	25.87	3.7		
cswec-ripple16-2000	6	17.03	5.6		
cswec-ripple16-2000	8	13.34	7.2		
cswec-ripple16-2000	10	11.14	8.6		
cswec-ripple16-2000	12	9.44	10.1		
cswec-ripple16-2000	14	8.37	11.4		

Table 6: Execution times for the circuit simulator with inputs "fiveripple" (with work factors 500 and 1000) and "ripple16" (with work factors 500 and 2000).

problem	nprocs	PC time (sec)	PC speedup	SGI time (sec)	SGI speedup
groebner-arnborg5	1	5.27	1.0	11.060	1.0
groebner-arnborg5	2	12.78	0.4	7.029	1.4
groebner-arnborg5	4	7.97	0.7	4.950	2.2
groebner-arnborg5	6	1.49	3.5	3.786	2.9
groebner-arnborg5	8	1.97	2.7	6.209	1.8
groebner-arnborg5	10	1.12	4.7	2.522	4.4
groebner-arnborg5	12	0.97	5.5	2.281	4.8
groebner-arnborg5	14	1.19	4.4	2.651	4.2
groebner-katsura4	1	1.21	1.0	2.968	1.0
groebner-katsura4	2	0.63	1.9	1.662	1.8
groebner-katsura4	4	0.39	3.2	1.191	2.5
groebner-katsura4	6	0.33	3.7	0.862	3.4
groebner-katsura4	8	0.29	4.2	0.684	4.3
groebner-katsura4	10	0.30	4.0	0.838	3.5
groebner-katsura4	12	0.28	4.4	0.708	4.2
groebner-katsura4	14	0.29	4.1	0.757	3.9
groebner-pavelle5	1	10.54	1.0	28.434	1.0
groebner-pavelle5	2	5.82	1.8	18.475	1.5
groebner-pavelle5	4	4.19	2.5	12.311	2.3
groebner-pavelle5	6	3.17	3.3	8.897	3.2
groebner-pavelle5	8	2.47	4.3	6.948	4.1
groebner-pavelle5	10	2.03	5.2	6.318	4.5
groebner-pavelle5	12	1.90	5.5	5.494	5.2
groebner-pavelle5	14	3.06	3.4	9.414	3.0

Table 7: Execution times for Groebner basis calculation. The arnborg5, katsura4, and pavelle5 are standard benchmarks for Groebner performance.

problem	nprocs	H (bytes)	S	PC $gH + LS$ (sec)	PC time (sec)	PC %
empic-small	2	2080208	240	0.19	10.83	1.8
empic-small	4	4835676	240	0.53	5.88	9.0
empic-small	6	5021720	240	0.63	4.18	15.1
empic-small	8	6200652	240	0.85	3.30	25.8
empic-small	10	5729256	240	0.85	2.90	29.3
empic-small	12	6441632	240	1.06	2.57	43.7
empic-small	14	6777272	240	1.33	2.34	56.8
msp-w100-sp-u10k	2	994152	896	0.14	1.96	7.1
msp-w100-sp-u10k	4	972956	626	0.21	1.08	19.4
msp-w100-sp-u10k	6	834128	506	0.24	0.95	25.3
msp-w100-sp-u10k	8	731388	426	0.26	0.87	29.9
msp-w100-sp-u10k	10	604468	446	0.31	0.79	39.2
msp-w100-sp-u10k	12	611668	366	0.32	0.70	45.7
msp-w100-sp-u10k	14	618868	366	0.39	0.71	54.9

problem	nprocs	H (bytes)	S	SGI $gH + LS$ (sec)	SGI time (sec)	SGI %
empic-small	2	2080208	240	0.30	24.15	1.2
empic-small	4	4835676	240	0.69	13.43	5.1
empic-small	6	5021720	240	0.75	9.85	7.6
empic-small	8	6200652	240	0.89	7.71	11.5
empic-small	10	5729256	240	0.84	6.79	12.4
empic-small	12	6441632	240	0.96	5.96	16.1
empic-small	14	6777272	240	1.03	6.11	16.9
msp-w100-sp-u10k	2	994152	896	0.16	6.04	2.6
msp-w100-sp-u10k	4	972956	626	0.16	3.27	4.9
msp-w100-sp-u10k	6	834128	506	0.14	2.73	5.1
msp-w100-sp-u10k	8	731388	426	0.12	2.20	5.5
msp-w100-sp-u10k	10	604468	446	0.11	1.88	5.9
msp-w100-sp-u10k	12	611668	366	0.11	1.59	6.9
msp-w100-sp-u10k	14	618868	366	0.13	1.68	7.7

Table 8: BSP communication and synchronization parameterization for "empic-small" and "msp-w100-sp-u10k." The cost model is used to estimate the percentage of execution time spent on communication and synchronization.