Programming Research Group

PRAM PROGRAMMING: THEORY VS. PRACTICE

A. V. Gerbessiotis, D. S. Lecomber, C. J. Siniolakis, and K. R. Sujithan

PRG-TR-21-97

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

PRAM Programming: Theory vs. Practice*

A. V. Gerbessiotis, D. S. Lecomber, C. J. Siniolakis, and K. R. Sujithan
Programming Research Group,
Oxford University Computing Laboratory, Oxford OX1 3QD, UK
{alexg,dsl,cs,krs}@comlab.ox.ac.uk

June 1997

Abstract

In this paper we investigate the practical viability of PRAM programming within the BSP framework. We argue that there is a necessity for PRAM computations in situations where the problem exhibits poor data locality. We introduce a C++ PRAM simulator that is built on top of the Oxford BSP Toolset, *BSPlib*, and provide a succinct PRAM language. Our approach achieves simplicity of programming over direct-mode BSP programming for reasonable overhead cost. We objectively compare optimised BSP algorithms with PRAM algorithms implemented with our C++ PRAM library and provide encouraging experimental results for the latter style of programming.

*The work of the first author was supported in part by EPSRC(UK) under grant GR/K16999, and the third author was supported by a Bodossaki Foundation Graduate Scholarship.

1 Introduction

Research in the theory of parallel computation has primarily focussed on parallel machines that communicate via shared memory, the so called *Parallel Random Access Machines* (PRAMs) [7]. A PRAM is an ideal parallel computer: a potentially unbounded set of processors sharing a global address space. The processors work synchronously and during each time step each processor either performs a computation or accesses a single data word from the global address space in unit time. Thus, the PRAM model abstracts parallelism by stripping away considerations such as communication latency, memory and network conflicts during routing, bandwidth of interconnection networks, memory management, and processor synchronization. Consequently, numerous PRAM algorithms have been proposed in the past several years, and a number of powerful techniques for designing such algorithms have been identified [15, 16].

Unfortunately, no realization of such shared memory models seems feasible in the foreseeable future; the architectural challenge of producing a genuinely scalable machine with unit communication cost has not yet been resolved. The *Bulk-Synchronous Parallel* (BSP) model has been introduced as a high-level abstraction of hardware with the purpose of allowing parallel programs to be scalable and run efficiently on diverse hardware platforms. The primary contribution of this paper is the unification of the BSP and PRAM models in a common programming environment. In particular, we study the viability of supporting PRAM computations on the BSP model.

2 The BSP Model

The *Bulk-Synchronous Parallel* (BSP) model of computation has been proposed by L.G. Valiant [28], as a unified framework for the design, analysis, and programming of general purpose parallel computing systems. It offers the prospect of achieving both scalable parallel performance and architecture independent parallel software, and provides a framework which permits the performance of parallel and distributed systems to be analyzed and predicted in a precise way. The term *bulk-synchronous* reflects the underlying position of the model between the two extremes, (i) entirely synchronous systems, and (ii) fully asynchronous systems. The BSP computer as described in [28] and further explored in [9, 19, 20] consists of the following three components: (i) a *collection of p processor/memory components* numbered $0, \ldots, p-1$, (ii) a *communication network* that can deliver messages point to point among the processors, and (iii) a *mechanism* for efficient *barrier synchronization* of all the processors.

The performance of any BSP computer can be characterised by the following three parameters: (i) p – the number of processors, (ii) g – the time required to realise h-relations in continuous message usage in gh time units [20], and (iii) l – the minimum elapsed time, measured in basic computational operations, between successive synchronization operations.

Computation on the BSP model proceeds in a succession of *supersteps*. During a superstep each processor is given a task to perform using data already available there locally before the start of the superstep. The task may include (i) computation steps on values held locally at the start of the superstep, (ii) message transmissions, and (iii) message receipts. A superstep can complete at any time after the first l time units. The time complexity of a superstep $\mathcal{S}$ in a BSP algorithm is defined as $\max\{l, x+gh\}$ time units, where x is the maximum number of basic computational operations executed by any processor during $\mathcal{S}$ and h is the maximum number of messages transmitted or received by any processor.

2.1 PRAM vs. BSP Programming

Two modes of programming were envisaged based on the BSP model: (i) *automatic-mode* where programs are written in a high-level language that hides low-level details such as memory allocation, and (ii) *direct-mode* where the programmer retains control of memory distribution.

Direct-mode programming has been successfully utilised for a multiplicity of problems [9, 20], and optimal results have been obtained for combinatorial and geometric problems, linear algebra and numerical analysis, data structures and databases [9, 4, 10, 11, 13, 19, 27, 26].

As noted in [9], the direct-mode of programming is advantageous in the following circumstances:

(1) where small multiplicative constant factors are important,

(2) where small problem instances can be run more efficiently in direct-mode (less slackness is required) than in automatic-mode.

(3) the underlying BSP machine has high g, and

(4) l is high for direct but not for automatic-mode for the problem instance in hand.

Direct-mode programming thus emphasises multiplicative constant factors close to one, i.e. *one-optimal* algorithms. In this paper we investigate the practicality of the alternative automatic-mode of programming, where, for the sake of simplicity, the programmer is relieved of explicit memory distribution.

Problems for which the direct-mode may possibly provide no better performance than automatic-mode are problems with poor locality such as symbolic problems, i.e., like list-ranking and sparse matrix computations. For such problems it is sufficient to employ automatic style BSP programming rather than direct style. A tool to support automatic-mode programming is therefore desirable.

3 Locality Analysis

In this section, we quantify the intuitive notions of communication complexity and locality and develop a sound basis for unifying the BSP and PRAM computations. In [15], the communication complexity of a PRAM algorithm is defined as the worst case bound on traffic between the shared memory and any processor executing that algorithm. This is inappropriate for BSP computations as it is applied over the algorithm as a whole and does not account for imbalance within communication steps. In [2] communication phases are considered; in each phase a single communication can be made by each processor to the shared memory; the total number of such phases constitutes the communication complexity. Accordingly, since this is essentially a h-relation idea, we adapt this concept to form a new definition of BSP communication complexity.

Definition 3.1 *The BSP communication complexity $M(\mathcal{A}, p)$ of a p-processor BSP* algorithm *A is the sum (over all supersteps) of h_i, where during superstep i algorithm A realises a h_i-relation.*

Definition 3.2 *The BSP communication complexity $M(\mathcal{Q}, p)$ of a problem $\mathcal{Q}$ is the minimum over the communication complexities $M(\mathcal{A}, p)$ of all p-processor BSP algorithms $\mathcal{A}$ solving $\mathcal{Q}$.*

A notion of locality is defined in [25] as follows. Consider a two processor network in which the data of a problem of size n is balanced. Let the communication complexity $\bar{M}(n)$ of a problem be defined as the minimum (over all algorithms that solve the problem) of the number of elements that must be communicated over a link between the two processors. If the work required by a work-optimal PRAM algorithm to solve the problem is $\bar{W}(n)$ then the locality is defined to be $\bar{L}(n) = \bar{W}(n)/\bar{M}(n)$. We generalise this approach to suit BSP computations by employing a p-processor abstraction instead of the two processor abstraction.

Definition 3.3 *Let $W(\mathcal{Q}, p)$ be the minimum work over all work-optimal BSP algorithms for a problem $\mathcal{Q}$. The locality $L(\mathcal{Q}, p)$ of a* problem *$\mathcal{Q}$ involving p-processors is $L(\mathcal{Q}, p) = W(\mathcal{Q}, p)/M(\mathcal{Q}, p)$.*

The following result is then obtained [17].

Theorem 3.1 *If $L(\mathcal{Q}, p) = \Theta(1)$ then simulating an optimal PRAM algorithm on the BSP model does not create additional communication traffic over a direct BSP algorithm, except for multiplicative constant factors.*

Sparse matrix multiplication, bounded integer sorting, connectivity of an n vertex graph with m edges, and evaluating arithmetical expressions have localities of $\Theta(1)$, $\Theta(1)$, $\Theta(\alpha(n, m))$ and $\Theta(1)$ respectively [25]. Here, $\alpha(n, m)$ is the inverse of the Ackerman function and is practically $\Theta(1)$ for our purposes. A facility to simulate such PRAM algorithms on the BSP is thus desirable.

4 PRAM Computations in BSP: Theory

In this work we adopt the PRAM simulations of [9] and discuss the practical implications of implementing these simulations (along with a programming environment) on the BSP model. We only distinguish between *Exclusive Read Exclusive Write* (EREW) and *Concurrent Read Concurrent Write* (CRCW) PRAMs (respectively CPRAMs and EPRAMs).

As noted in [9], given sufficient parallel slackness, we can design PRAM simulations on the BSP model that are one-optimal in communication and optimal to within small multiplicative constant factors in computation. By a PRAM simulation being one-optimal in communication we mean that $T(n)$ read/write operations of a PRAM algorithm can be simulated in $2(1+o(1))gT(n)/p$ time units on a p-processor BSP machine [9]. The $2g$ factor is necessary since each read operation is implemented as a message to a distant memory block followed by a return message, and time g is charged per message.

Theorem 4.1 ([9]) *Let ω_p be any function of p such that $\omega_p \to \infty$ as $p \to \infty$. Then the following amounts of slackness are sufficient for simulating any one step of an EPRAM or CPRAM algorithm on a BSP machine in one-optimal time for communication (and constant factor optimal time in local operations if $g = O(1)$):*

(1) $(\omega_p\, p\, \log p)$-EPRAM on p-BSP,

(2) $(\omega_p\, p^2\, \log p)$-CPRAM on p-BSP.

Remark. The expressions before the hyphens denote the number of processors; $\omega_p\, \log p$ and $\omega_p\, p\, \log p$ denote the *parallel slackness* required for the EPRAM and CPRAM simulations respectively.

In order to achieve efficiency, in our implementation, we utilise hashing to map the address space across the distributed memory. In [8] it is established that by employing universal hashing [5], behaviour similar to the one achieved under the assumption of perfect hashing can be attained by a small class of hash functions. In particular, it is shown that the simulations implied in theorem 4.1 can be effected by employing two uniformly at random selected hash functions h_1 and h_2 from some high performance universal class $H_{p,n} \subset \{h : \{0,\ldots,n-1\} \to \{0,\ldots,p-1\}\}$ [6], where $\{0,\ldots,n-1\}$ denotes the set of memory cells employed by the underlying simulated PRAM algorithm, and $\{0,\ldots,p-1\}$ denotes the set of memory modules (processors) of the simulating BSP machine.

For the purpose of our experimental evaluation and in order to maintain *space* and *time* efficiency we utilise linear hash functions, i.e., $h(x) = ax \bmod m$, for a and m co-prime. The first $\log p$ bits of $h(x)$ represent the physical processor to which the address x is mapped and the least significant bits specify the location of the data on that processor. Hash function h is bijective from $\{0,\ldots,m-1\}$ onto itself and this allows the global space for a PRAM memory of n words to be $m = \Theta(n)$. This fact significantly avoids the need for secondary hashing for collision resolution. Although, there are many situations, theoretically, in which this class could result in high module congestion, in [1, 24] it is observed that linear hash functions perform well in practice. In particular, choosing $m = 2^k$, where $2^{k-1} < n \le 2^k$, simplifies calculation of h to one multiplication followed by a bitwise-AND. In addition, the space used globally to store a data set of size n is bounded above by $2n$. The simulation algorithm (EPRAM and CPRAM) is outlined next.

Algorithm BSP-PRAM

Superstep 1 :

- Locally sort the memory requests on each processor using a linear time integer sorting algorithm.
- For each multiple requests to the same memory location select a representative of the requests made to that address.
- Send the representatives to the processor(s) determined by the hash function.

Superstep 2 :

- On each processor, process the requests received. For writes this involves merely updating the value in local memory, for reads sending back the locally held value to the requesting processor.

Superstep 3 :

- Process the responses, duplicating the values where there were multiple read requests to the same location.

This algorithm works for both EPRAMs and CPRAMs, although the sorting stage is redundant, and therefore, not employed for the EPRAM model. We have implemented this algorithm on top of *BSPlib* and developed a PRAM language to facilitate easy implementation of PRAM programs [17].

5 PRAM Computations in BSP: Practice

We have developed a C++ class for managing the distributed memory of a BSP computer. A single class – the BSPArray – provides all the necessary functionality for the system. Parallelism is provided by the underlying *BSPlib* [14]. We have also implemented a language extension to ease the implementation of PRAM algorithms. A simple prototype C++ PRAM class library built on top of an earlier BSP library [23] was presented in [3].

5.1 The C++ PRAM Class Library

The C++ library we have developed is a flexible tool for realizing shared-memory arrays of polymorphic data type on BSP machines. By employing polymorphism we were able to address the containment of any homogeneous collection of data elements (array). The two constituent parts of an array are its placement and its elements. In our implementation, placement is handled by a view function, which determines the location of each index; in the PRAM context the view is our hash function.

C++ allows the redefinition of array-indexing to a purpose-built procedure without the need of writing a compiler. Reading and writing of elements results in BSP semantics being applied to the assignment operator: access is not guaranteed to happen until the next barrier synchronization has occurred. In the context of reading, the remotely held value cannot be used until the next synchronization point. Effectively all accesses are queued locally with the details of source and destination addresses until the system can handle them; such a decoupling of communication and synchronization is at the core of BSP and has been found to result in efficient algorithms [9].

For example, the following fragment of code reads the i-th entry of shared-memory array x and places it in local variable j on physical processor 0. After the implicit `sync` command, the value of variable j is the value of $x[i]$. Local variable j is of a new type `Slow<int>` reflecting the fact that it obtains its new value after the synchronization point. Variable j can be used as an ordinary integer in the next superstep by using the built-in type casting which we provide with the `Slow` class.

```
if (BSPme == 0) j = x[i];
x.sync();
```

From this basic class, the EPRAM and CPRAM memory classes are derived. These supply the hashing function, and in the latter handle combining of messages during synchronization as given by the algorithm. Without further embellishments, these classes may be used to provide PRAM-style shared memory in the standard BSP environment. We have described how PRAM memory is simulated and now proceed to describing our efficient simulation of PRAM processes.

5.2 Language Extensions

We have designed a concise macro extension to C++ to create a PRAM abstraction with many of the capabilities that PRAM-specific languages contain, e.g., FORK95. Our library can be integrated with standard BSP programs in a simple and consistent manner. We introduce only three new constructs.

- `PRAM_on (`*expression*`)` *statement* – Initialise a simulation of *expression* processes for the scope of *statement*.
- `PRAM` *statement* – Carry out *statement* on each of the virtual processes.
- `PRAM_if (`*expression*`)` $statement_1$ [`PRAM_else` $statement_2$]$_{opt}$ – Select those virtual processes for which *expression* is true and for the scope of $statement_1$ use just these. If a `PRAM_else` appears then $statement_2$ is executed with the remaining active processes.

A *statement* is any valid C++ statement, including further PRAM macros, and *expression* is a valid integer expression. We permit nesting of `PRAM_on` and `PRAM_if` statements but not of the `PRAM` which semantically must execute pure C++ statements.

The `PRAM_on` construct allots virtual processes to physical BSP processors. If appropriate, the old distribution is restored after the completion of the `PRAM_on` construct. The construct assigns a unique constant identifier (`PID`) to each of the virtual processes and evenly distributes these across the physical machine. The mapping of the virtual processes to physical processes is only modified by this construct. This allows processes to exploit the local memory of BSP machines for data caching and the avoidance of unnecessary shared-memory reads. In conjunction with this, the following additional variables are also defined.

- `PID` – globally unique virtual process identifier in $[0, \ldots, n-1]$, where n is the number of virtual processes.
- `PRAM_slack` – the number of active processes held on the local BSP processor (the sum of `PRAM_slack` over all physical processors is therefore n).
- `PIDl` – locally unique process identifier in $[0, \ldots, \texttt{PRAM\_slack} - 1]$.

The `PRAM_slack` and `PIDl` values are set for each virtual process at each `PRAM_on` and `PRAM_if` statement and remain constant for each `PRAM` statement. This allows efficient use of local memory on the physical BSP machine.

The `PRAM` construct initiates and supports the actual computation by executing *statement* on all virtual processes. The statement executed can be any native C++ statement based on locally held data (e.g. local procedure call), compound statements or shared-memory accesses, however, it must not involve BSP synchronization or further PRAM macros. The statement can depend on `PID`, `PRAM_slack` and `PIDl` if necessary.

Our alternation construct allows statements to be executed by processes that satisfy a predicate. The predicate may involve `PID` and any locally held variables. These statements do not invoke individual `PRAM` statements but they do change the set of active processes to the subset of those currently active which satisfy and fail the predicate respectively. The true/false branches are defined semantically to execute concurrently and internally synchronously (currently the system executes them in sequence). The old processor distribution is restored when both branches have finished and the next instruction is then executed (the following code illustrates a sample C++ PRAM matrix-multiplication program).

```
void Matrix_Mult (CRCW<double>& A, CRCW<double>& B, CRCW<double>& C, int n)
{
   PRAM_on (n * n) { Double AL[n * n], BL[n * n]; int i, j, k ; double t ;
   PRAM {                                   // First parallel step
        i = PID / n ; k = PID % n ;         // Row and column to calculate
        for (j = 0 ; j < n ; j++) {
            AL[i * n + j] = A[i * n + j] ;  // Read to local memory
            BL[j * n + k] = B[j * n + k] ;
        }
   }
   A.sync() ; B.sync();
   PRAM {                                   // Second parallel step
        i = PID / n ; k = PID % n ; t = 0 ;
        for (j = 0 ; j < n ; j++) {
            t += AL[i * n + j] * BL[j * n + k] ;
```

```
        }
        C[PID] = t ;
    }
    C.sync() ;
    }
}
```

Our PRAM simulation has a structured nature – all processors (virtual and physical) finish simultaneously and resume together the next block of a program with restored virtual processes sets if appropriate. Consequently, during the execution of a top-level PRAM simulation, the program state transformation can be specified; the end of the simulation can be followed by a barrier synchronization marking its completion and the limit of its influence. Therefore, we consider such subprograms as sequences of supersteps and reason about their external behaviour in the global single-threaded BSP manner.

The structure allows procedures containing PRAM simulations to be used in conventional BSP programs. BSP procedures can be called from within a `PRAM_on` block (or procedure) and vice-versa. Shared memory objects are conventional C++ objects. Virtual parallelism and supersteps can be handled within such procedures. By allowing a PRAM procedure to modify slackness or relabel the `PID`s assigned to physical processors with the `PRAM_on` construct, for the duration of its execution, it sets its own degree of parallelism. This allows a PRAM procedure to be unaffected by the number of processes that do not participate. A procedure need not inherit complex subsets of processes: PRAM algorithms can be written easier for blocks of processes such as $[0, \ldots, n-1]$. Varying slackness enhances modularity and reusability. The only "constant" is the data; it persists for the whole of a scope defined by conventional C++ scoping rules.

The system allows libraries of PRAM programs to be constructed to the fullest extent of C++ including polymorphic class libraries. We allow the definition of new classes that can then be placed in shared memory arrays. New classes can also be derived from the EPRAM and CPRAM classes. We have not yet considered further implications for object-oriented or object-based programming that our language and library raises. Other extensions based on the underlying distributed array package are under development.

6 Experimental Results

We present results on three computational problems that exhibit different levels of locality: list ranking, matrix multiplication and bitonic sorting. The list ranking problem has locality $\Theta(1)$ [17]; in contrast, sorting and matrix multiplication are computation bound, i.e. $\Omega(1)$ locality. Matrix multiplication of two $n \times n$ matrices can be achieved by a direct-mode BSP algorithm in optimal $\Theta(n^3/p)$ and $\Theta(n^2/p^{2/3})$ computation and communication time respectively [20]. Thus, the locality of matrix multiplication is $\Theta(n/p^{1/3})$. Finally, optimal BSP sorting has been shown to have locality $\Theta(\lg(n/p))$ [13]; by contrast, in [12], it is shown that bitonic sorting has locality $\Theta(1 + (\lg n/p)/\lg^2 p)$.

We have developed efficient BSP implementations of these algorithms on top of *BSPlib*. We have also implemented the corresponding PRAM algorithms on our C++

PRAM library. For list ranking we coded the EREW random-mate algorithm of [23], that employs $O(n/\lg n)$ processes. Matrix multiplication employs a straightforward n-process CREW algorithm. For sorting we use the non-optimal bitonic sorting algorithm [15] – favouring this over more complicated optimal algorithms. As this algorithm is not efficient, we improved its performance by implementing a C++ class that replaces comparators by multi-way mergers [12]. This approach demonstrates the benefits of our C++ PRAM library polymorphic capabilities, i.e., we were able to replace the comparators with mergers thus allowing blocks of sorted arrays to be merged, and still use the original program.

In the tables and graphs to follow the following remarks apply.

- CPU times are given in seconds on an SGI Power Challenge shared memory system, consisting of four 64-bit Streaming Superscalar TFP processors, for a total capacity of 1.2Gflops peak with 512MBytes RAM, and running the `IRIX 6.2 IP21` operating system.
- All direct-mode BSP algorithms are written in C and linked with the Oxford BSP Toolset, *BSPlib*. The code was compiled using the native `cc` compiler with option `-O3`.
- All PRAM algorithms are written in C++ and linked with the Oxford BSP Toolset, *BSPlib*, and our C++ PRAM library. The code was compiled using the native `CC` compiler with option `-O3`.
- The high-precision real-time clock function `bsp_time` [14] (wall clock time) has been employed for timings.
- For each test problem instance, we performed a series of 8 experiments, and the running times indicated are the average of those 8 experiments.

	1 Physical Processor		4 Physical Processors	
List Size	Direct Time (s)	PRAM Time (s)	Direct Time (s)	PRAM Time (s)
8192	0.217	0.71	0.103	0.28
32768	0.891	2.87	0.378	0.96
131072	5.192	12.90	1.655	3.67
524288	35.862	61.77	13.394	16.66

Table 1: List Ranking on SGI Power Challenge.

The list ranking problem is inherently non-local and therefore results in complicated and irregular data access patterns. This situation is handled effectively by our system. As exhibited in table 1 the slowdown of the PRAM simulation over the direct-mode BSP implementation is in the region of 1-3. We note that the slowdown factor increases as the problem size decreases and this may be attributed to the C++ PRAM library overheads.

	1 Physical Processor		4 Physical Processors	
Matrix Size	Direct Time (s)	PRAM Time (s)	Direct Time (s)	PRAM Time (s)
64 x 64	0.008	0.21	0.003	0.07
128 x 128	0.053	1.30	0.015	0.41
256 x 256	0.381	10.22	0.103	2.37
512 x 512	3.005	79.23	0.785	20.10

Table 2: Matrix Multiplication on SGI Power Challenge.

Matrix multiplication is computation bound and its structured communication patterns allow for efficient direct-mode BSP implementations. Moreover, the structured computation patterns fully utilise the first- and second-level cache of the system, and therefore, can be advantageously exploited in direct-mode BSP implementations. On the other hand, the underlying hashing scheme of our C++ PRAM library destroys locality and this is reflected in the slowdown of the simulation (refer to table 2 and figure 1). The PRAM algorithm we implemented is CREW in nature and therefore sustains considerable overheads due to combining of memory requests (superstep 1 of algorithm BSP-PRAM) – the reduction of such overheads is a task on which our efforts are now focussed. An alternative way of alleviating the 20-30 slowdown factor of the simulation is by increasing the data granularity, as we do for the sorting problem.

	1 Physical Processor			4 Physical Processors		
Size	Direct Time(s)	Virtual Processes	PRAM Time (s)	Direct Time(s)	Virtual Processes	PRAM Time (s)
4096	0.011	256	0.31	0.004	256	0.13
		4	0.027		4	0.012
16384	0.051	1024	1.90	0.073	1024	0.70
		4	0.11		4	0.085
65536	0.310	32	1.18	0.132	32	0.42
		4	0.52		4	0.18
262144	1.200	128	13.41	0.344	128	3.87
		4	2.79		4	0.90

Table 3: Bitonic Sorting on SGI Power Challenge.

Bitonic sorting is structured in nature, yet it exhibits a low degree of locality. The PRAM algorithm implementation handles effectively this situation by employing block structures instead of single word elements, thus reducing the slowdown factor from 10-30 (large number of virtual processes) to 0.7-3 (small number of virtual processes) (refer to table 3 and figure 1). Optimal performance, almost equaling that of direct-mode programming is obtained when virtual to physical slackness approaches unity. In this situation the PRAM algorithm is equivalent to the direct-mode BSP algorithm – except for the assumption that all data is non-local. This example shows the flexibility of our system – we were able to use a simple PRAM algorithm to implement a portable semi-automatic BSP sorting algorithm.

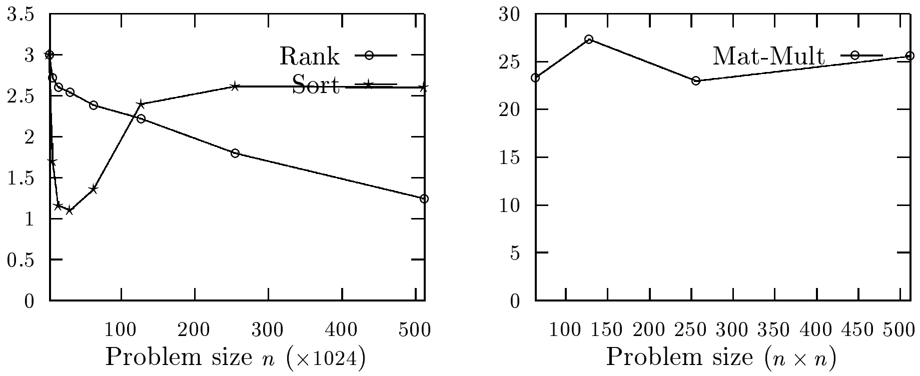

Figure 1: Slow-down results for list ranking, bitonic sorting and matrix multiplication.

7 Conclusion and Future Directions

We have presented a practical study of PRAM simulations on the BSP model and shown that for some problems the cost of simulating PRAM algorithms on BSP machines is comparable (to within small multiplicative constant factors) to that of direct-mode BSP solutions. As theory suggests, low locality problems such as, list ranking and bitonic sorting, performed well on our system as compared to direct-mode implementations. Our PRAM programming environment is easy to program with, and can be effectively used in conjunction with, other BSP programs. The performance of our PRAM algorithms is competitive to that of the corresponding direct-mode BSP algorithms, and therefore, justifies further development of these techniques.

Future work will focus on reducing the constant factors of the simulation even further. Our automatic-mode programs were written in C++, but the direct-mode programs were coded in C which is known to result in faster code. Without C++, however, many of the advanced features of the PRAM language would need to be replaced by more cumbersome constructs – alternatively we could develop a compiler that handles shared memory efficiently during compilation. The experimental results we have presented show that, even with the associated slowdown of the C++ language, the overheads are competitively low, indicating the practical viability of PRAM programming within the BSP framework.

Acknowledgements. We thank the members of the Oxford BSP research group for several helpful discussions. In particular, we thank Jon Hill for his valuable assistance with *BSPlib* throughout this work.

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, and D. Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756-762, 1993.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71:3-28, 1990.

[3] A. G. Alexandrakis, A. V. Gerbessiotis, D. S. Lecomber and C. J. Siniolakis. Bandwidth, space and computation efficient PRAM programming: The BSP approach. In *Proceedings of the SUPEUR'96 Conference*, Krakow, Poland, September 1996.

[4] A. Baumker, W. Dittrich, F. Meyer auf der Heide and I. Rieping. Realistic parallel algorithms: Priority queue pperations and selection for the BSP* model. In *Proceedings of EUROPAR'96*, LNCS, Volume 1124, Springer Verlag, August 1996.

[5] J. L. Carter and M. N. Wegman. Universal classes of hash functions. In *Proceedings of the* 9*-th ACM STOC*, pp. 106-112, 1977.

[6] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of International Colloquium on Automata, Languages and Programming*, pp. 6-19, LNCS, Volume 443, Springer Verlag, 1990.

[7] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10-th ACM STOC,* pp. 114-118, 1978.

[8] A. V. Gerbessiotis. Topics in parallel and distributed computing. PhD Thesis (Technical Report TR-03-93), Aiken Computing Laboratory, Harvard University, 1993.

[9] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251-267, 1994.

[10] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the 8-th ACM SPAA*, June 1996.

[11] A. V. Gerbessiotis and C. J. Siniolakis. Communication efficient data structures on the BSP model with applications in computational geometry. In *Proceedings of EUROPAR'96*, LNCS, Volume 1124, Springer Verlag, August 1996.

[12] A. V. Gerbessiotis and C. J. Siniolakis. Primitive Operations on the BSP model. Technical Report PRG-TR-23-96, Computing Laboratory, Oxford University, October 1996.

[13] M. T. Goodrich. Communication-efficient parallel sorting. In *Proceedings of the 26-th ACM STOC*, 1996.

[14] M. W. Goudreau, J. M. D. Hill, K. Lang, W. F. McColl, S. B. Rao, D. C. Stefanescu, T. Suel, T. Tsantilas. A proposal for the BSP worldwide standard library (preliminary version). Technical Report, Oxford University Computing Laboratory, April 1996 (see `www.bsp-worldwide.org` for details).

[15] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[16] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science,* (van Leeuwen, J., ed.), North Holland, 1990.

[17] D. S. Lecomber and K. R. Sujithan. Transgressing the boundaries: Unified scalable parallel programming. Technical Report PRG-TR-20-96, Oxford University Computing Laboratory, November 1996.

[18] D. S. Lecomber and K. R. Sujithan. Architecture-independent locality analysis and efficient PRAM simulations. In *Proceedings of HPCN'97* (to appear).

[19] W. F. McColl. General purpose parallel computing. In *Lectures on parallel computation,* (A. Gibbons and P. Spirakis, eds.), Cambridge University Press, 1993.

[20] W. F. McColl. Scalable parallel computing: A grand unified theory and its practical development. In *Proceedings of IFIP World Congress*, 1:539-546, Hamburg, August 1994.

[21] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulation of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339-374, 1984.

[22] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *Proceedings of the 26-th IEEE Symposium on Foundations of Computer Science*, pp. 478-489, 1985.

[23] R. Miller. A library for bulk-synchronous parallel programming. In *Proceedings of the BCS Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, 1993.

[24] A. G. Ranade, S. Bhatt, and S. Johnson. The Fluent abstract machine. In *Proceedings of 5-th MIT Conference on Advanced Research in VLSI*, pages 71-93. MIT Press, 1988.

[25] A. G. Ranade. A framework for analyzing locality and portability issues in parallel computing. In *Parallel architectures and their efficient use*, First Heinz Nixdorf Symposium, LNCS, Volume 678, Springer-Verlag, 1993.

[26] C. J. Siniolakis. Bulk-synchronous parallel algorithms in computational geometry. Technical Report PRG-TR-10-96, Oxford University Computing Laboratory, May 1996.

[27] K. R. Sujithan. Towards a scalable parallel object database – The bulk-synchronous parallel approach. Technical Report PRG-TR-17-96, Oxford University Computing Laboratory, August 1996.

[28] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103-111, 1990.