PRAM Programming: In Theory and In Practice

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

Programming Research Group,

Oxford University Computing Laboratory, Oxford OX1 3QD, UK

{dsl,cs,krs}@comlab.ox.ac.uk

Abstract

That the influence of PRAM model [FW78] is ubiquitous in parallel algorithm design is as clear as the fact that it is technologically infeasible for the forseeable future. The current generation of parallel hardware prominently features distributed memory and high-performance interconnection networks – very much the antithesis of the shared-memory required for the PRAM model. It has been shown that, in spite of communication costs, for some problems very fast parallel algorithms are available for distributed-memory machines – from embarassingly parallel problems [Fox95] to sorting and numerical analysis [GV94]. In contrast it is known that for other classes of problem PRAM-style shared-memory simulation on a distributed-memory machine can – in theory – produce solutions of comparable performance to the best possible for such architectures.

The *Bulk Synchronous Parallel* (BSP) model [Val90] accurately represents most parallel machines – theoretical and actual – in an execution and cost model. We introduce a scalable portable PRAM realization appropriate for BSP computers and a methodology for usage. Our system is fast and built upon the familiar sequential C++ coupled with the new standard BSP library [GHL+96] of parallel computation and communication primitives. It is portable to and predictable on a vast number of parallel computers including workstation clusters, a 256 processor Cray T3D, an 8 node IBM SP/2 and a 4 node shared-memory SGI Power Challenge machines. Our approach achieves simplicity of programming over direct-mode BSP programming for reasonable overhead cost. We objectively compare optimized BSP and PRAM algorithms implemented with our C++ PRAM library and provide encouraging experimental results for our new style of programming.

1 Introduction

The vast majority of *theoretical* parallel algorithm design in the last twenty years has primarily been targetted at large scalable machines that communicate via shared memory, the so called *Parallel Random Access Machines*(PRAMs) [FW78]. The 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. PRAMs may be subdivided according to the memory capabilities. The EREW PRAM (or EPRAM) is a processor in which no individual memory location is accessible by more than one processor in the same timestep. The arbitrary CRCW PRAM (or CPRAM) allows multiple writes and reads of the same location in the same timestep.

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. The PRAM captures our intuitive perception of what a perfect parallel machine should be – allowing a concentration on the pure theoretical complexity of a problem without concern for pragmatic complications. A vast collection of algorithms and techniques are known for the model [JáJ92, KR90].

In contrast, *practical* parallel programming is currently under the domination of distributed memory computers. These machines take many forms, from workstation clusters to large high-performance machines (for example the Cray T3D or IBM SP/2). Vendors of high-performance parallel computers are clearly of the opinion that systems of fast sequential processors with large local memory coupled to fast communication networks are the future. With industrial producers aiming in this direction it is reasonable to assume that, at least for the forseeable future, distributed memory will continue to dominate. No realization of theoretical shared memory models seems feasible at present; the architectural challenge of producing a genuinely scalable machine with unit communication cost has not yet been resolved. Tremendous effort has been expended on producing a fully scalable PRAM [FKW96, ADK+93] with, so far, only modest success.

It is increasingly accepted that a prerequisite for portable and scalable parallel computing is a simple accurate method of performance prediction. Realistic parallel machines are very diverse which complicates this goal. Recently a plethora of models [LMR95] have been introduced as bridging models to unify these machines with programming and cost methodologies. The *Bulk-Synchronous Parallel*(BSP) model [Val90, RPL96] is one such model and appears to be the most common. The model is a high-level abstraction of hardware for the purpose of allowing parallel programs to be scalable and run efficiently on diverse hardware platforms. A library implementation, based on a succint collection of primitives, of BSP was introduced in [Mil93, RPL96]. Its successor, *BSPlib* [GHL+96], is widely available for many systems. BSP may also be considered as a simplified and more intellectually manageable approach than the highly flexible MPI [SOHL+95, CFT+94] or PVM; *BSPlib* obtains performance equalling both these systems.

Theory provides a motivation for emulating PRAM memory in BSP for problems that have little *locality* or, alternatively, are difficult to program due to complex communication patterns. We introduce a fast simulation of PRAM machines to fill the practical void. Obviously emulation will have slow-downs related to the performance of the communication network and we show how these are quantifiable in the BSP cost model. For the problems described such slow-down is optimal for BSP machines and consequently relieving the programmer of memory allocation comes - in theory - without penalty. We present a practical investigation of this claim. This paper contributes by unifying the BSP and PRAM models in a common programming environment. We introduce a powerful scalable class library simulating the different PRAM memory models on any BSP machine; we also describe an extension to the C++ language to facilitate its use.

2 The BSP Model

The *Bulk-Synchronous Parallel* (BSP) model of computation has been proposed in [Val90] 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. Predictability is an important issue in parallel computing and BSP is designed with this in mind. In contrast, estimating run time in many other models of parallel computing is difficult.

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 [Val90] and further explored in [GV94, McC93, McC95] consists of the following three components: (i) a *collection of p*

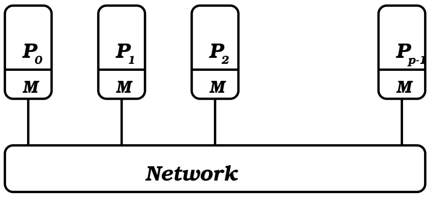

Figure 1: The BSP Computer

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.

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. The performance of any BSP computer can be characterised by the following three parameters, where time is measured in local flops lost:

- p – the number of processors,
- g – where the time required to realise *h-relations*[1] in continuous message usage is gh time units, and
- l – the minimum time between successive synchronization operations.

The constraints of bulk synchronisation and emphasis on message balance are sufficient for highly predictable running times. 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 or equivalently the h-relation realized by the superstep.

2.1 PRAM vs. BSP Programming

Two modes of programming were envisaged based on the BSP model: (i) *direct-mode* where the programmer retains control of memory distribution [Val90], and (ii) *automatic-mode* where programs are written in a high-level language that hides low-level details such as memory allocation.

[1] A h-relation is a message pattern in which no processor receives more, nor sends more, than h words

Direct-mode programming has been successfully utilised for a multiplicity of problems [GV94, McC95], and optimal results have been obtained for combinatorial and geometric problems, linear algebra and numerical analysis, data structures and databases [GV94, GS96b, GS96a, Goo96, McC93, Suj96, Sin96]. Some of these results have been obtained using the Oxford BSP Toolset, *BSPlib*, which is a highly efficient realization of the BSP model [GHL$^+$96]. It is also possible to implement BSP using subsets of other parallel libraries, for example, MPI [SOHL$^+$95, CFT$^+$94]. *BSPlib* has the advantage over MPI of being extremely simple. It is based on a small package of a dozen communication primitives and as such provides a good introduction to high performance parallel library based programming.

[GV94] suggests that 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 and communication. In contrast to proposition (3) above, we propose that there are problems for which – regardless of g – the direct-mode may possibly provide no better performance than automatic-mode. Such problems have poor *locality*. Typical examples include symbolic problems, i.e., like list-ranking and sparse matrix computations. Theory shows that it is sufficient to employ automatic style BSP programming rather than direct style. A tool to support automatic-mode programming is therefore desirable and this is the problem our package answers.

In particular it is clear that a cross-paradigm approach is required to produce some large systems that selects the appropriate approach for each subproblem arising. This is essential to modularity. We integrate the message-passing, the shared-memory and also data-parallel frameworks. The importance of paradigm unification has been realized in other publications (notably [CF96, GC92]). The BSP semantic model makes this unification less complex than such papers have previously expected.

3 PRAM Computations in BSP: Theory

Theoretical simulations of PRAMs on more realistic parallel machines are well documented (see for example [Har94, KLM92, Val90, MV84]). [KLM92, CMS95] and many others concern *Distributed Memory Machines* (DMMs) which differ from BSP machines in that they can limit the number, c, of communications served by any one machine in a time step. Even if $h > c$ messages are sent to any one processor then c are served and the cost is merely c globally. The DMM model is seen as realistic due to the possibility of optimal communication where a fixed constant number of cells have concurrent read/write capability. However this communication model is harder to simulate in BSP than the EREW PRAM model – which therefore rules out using DMM simulations as an intermediate level between BSP and PRAM. We use the following definition of [Val90] to discuss BSP results.

Definition 3.1 *The* slackness *of a simulation of an n processor by a p processor computer is defined to be the ratio n/p.*

To distinguish between the n processors of a simulated machine and p of the actual, we will often use the term virtual processor and physical processor respectively.

It is also important to quantify the efficiency of a simulation. We use the notion of slowdown for this task.

Definition 3.2 *If machine $\mathcal{A}$ machine is simulated by machine $\mathcal{B}$,* slowdown (or delay) *is the number timesteps $\mathcal{B}$ requires to simulate a timestep of $\mathcal{A}$*

Simulation of the n-processor PRAM on n-processor realistic machines using totally randomised memory is known to introduce slowdown of $\frac{\log n}{\log \log n}(1+g)$ in the BSP model. However, by increasing slackness it is found that simulations exist for which the slowdown is the optimal $\Theta((1+g)n/p)$.

For the DMM model simulation of PRAMs has achieved very low bounds on the amount of slackness required for optimal simulation. Notably [CMS95] achieve simulation of an $n \log\log\log n \log^* n$-EREW with delay of $O(\log\log\log n \log^* n)$ on an n-processor DMM. The techniques obtaining this result unfortunately do not adapt to BSP. Even on the DMM model the constant factors appear to be prohibitive in these powerful simulations – although to our knowledge no precise quantification has ever been undertaken of these constants. [LRGD97] imply this to be true of all simulations and consequently develop an approach which produces efficient reads at the cost of expensive writes and large space usage. The simulation of [GV94] shows that such a generalisation is too strong, as they develop a randomised approach in which the overhead is merely two communications and a small number of computations for both read and write operations at the expense of slackness of $\omega(\log n)$ for the EREW model. Fast simulations are also obtained on a butterfly network in [ADK$^+$93] - again using logarithmic slackness.

Our simulation is an improvement on that reported in [LRGD97] in which p balanced write operations cost pg compared to g (with high probability) in [GV94]. Their original approach is feasible for only a small class of algorithms, namely that in which the ratio of read operations to writes exceeds p to 1. [MV84] note that in practice the read/write ratio of typical PRAM programs is around 8 to 1. The approach of [LRGD97] also has global memory requirement of $\Omega(np)$ where n is the size of the PRAM memory being simulated. Such space factors are becoming increasingly dominant in parallel programming where minimizing memory use eliminates costly cache misses and virtual (disk based) memory. Memory usage in [ADK$^+$93] is, like ours, optimal at $\Theta(n)$. Methods similar to our own appear on transputers in [CM96]

We therefore state the following:

Theorem 3.1 *[GV94] 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 EREW PRAM or CRCW PRAM algorithm on a BSP machine in one-optimal[2] time for communication (and constant factor optimal time in local operations if $g = O(1)$):*

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

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

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

The bound of slackness for CRCW can be further improved for a penalty in the constant factors of simulation using a different algorithm (see [Val90]) to the one we explore.

[2]By 1-optimal we mean that the constant multiplicative factors involved in the communication and computation overheads are g and 1 respectively

Our simulation utilises a constant-time perfect hash function to map the address space across the distributed memory randomly. In [Ger93] it is established that the hash functions described in [DM90] suffice. 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. Although, there are many situations theoretically in which this class could result in high module congestion, in [ADK$^+$93, RBJ88] it is observed that linear hash functions perform well in practice. 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. In particular, choosing $m = 2^k$, where $2^{k-1} < n \leq 2^k$, 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 (EREW and CRCW PRAM) is outlined next.

Algorithm BSP-PRAM

Superstep 1 :

- Locally eliminate duplicate memory requests in linear time - using, for example, a local hash table or a linear integer sort routine. For each of the 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 global hash function h.

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 EREW and CRCW PRAMs – although the duplicate removal stage is redundant and therefore not employed for the EREW model.

We have implemented this algorithm on top of *BSPlib* and developed a PRAM language to facilitate easy implementation of PRAM programs.

4 Locality Analysis

Having established the bounds achievable for PRAM simulation, we proceed to analyse those problems for which that approach is viable. We quantify intuitive notions of communication complexity and locality and develop a sound basis for unifying the BSP and PRAM computations. In [JáJ92], 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 [ACS90] 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 4.1 *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 4.2 *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 [Ran93] 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 4.3 *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 [LS96].

Theorem 4.1 *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 [Ran93]. 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. On the other hand, there are many problems which do exhibit locality, for example comparison sorting, dense linear algebra problems and some computational geometry problems [GV94, Sin96]. Some of these problems are neither irregular nor dynamic, and therefore, as shown in the following sections they can – be simulated efficiently.

5 PRAM Computations in BSP: Implementation

We have developed a C++ class for managing the distributed memory of a BSP computer to replace explicit fetch and store of remote data by a more modern approach. A single class – the `BSPArray` – provides a base for the system and from this class we derive the `EREW` and `CRCW` memory classes. A static parallel environment is provided by the underlying *BSPlib* [GHL$^+$96] on which we implement our dynamic PRAM parallelism. We provide a language extension to ease the implementation of PRAM algorithms which is similar in nature to the other *PRAM-oriented* [LRGD97] languages of 11 [LRGD97] and FORK95 [HSS94, KT96] but we additionally introduce generic data types and other C++ benefits. A simpler prototype C++ PRAM class library, without language embellishments and built on top of an earlier BSP library [RPL96, Mil93], was presented in [AGLS96].

5.1 The C++ PRAM Class Library

The C++ base library we have developed is a flexible tool for realizing distributed arrays, with placement functions, 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 and assignment for a type to purpose-built procedures – therefore without the need for writing a compiler we are able to change the way data is accessed for our shared-memory class and introduce a BSP approach appropriate to our needs.

In the context of reading and writing global memory, we apply BSP semantics to the assignment operator: access is not guaranteed to happen until the next barrier synchronization has occurred. In reading, the remotely held value cannot be used until the next synchronization point. Effectively all accesses are queued locally (using the `view`) with the details of source and destination addresses until the system can handle them. Decoupling of communication and synchronization is at the core of BSP and has been found to result in efficient algorithms [GV94]. The system expects explicit synchronisation invocation.

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();
```

The basic array class can be used without CRCW/EREW intentions as a simple interface to a data-placement which avoids the more basic untyped `bspstore` and `bspfetch` of the BSP libraries. The `view` function can be used to place data in any appropriate distribution.

For our purpose we derive EREW PRAM and CRCW PRAM memory classes. These supply the random hash function, and in the latter handle combining of messages during synchronization as given by the algorithm BSP-PRAM. 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 most of the capabilities that PRAM-specific languages contain. 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 space 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 `PID1` 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, an approach also adopted in [LRGD97]). The old processor distribution is restored when both branches have finished and the next instruction is then executed.

Local memory for each virtual processor can be defined using another class we introduce which allots the correct space on each physical processor – thus providing space efficiency. The class is `Local<T>` and each declaration of an object of this class introduces `PRAM_slack` elements on each physical processor. We can also opt to just use the physical BSP memory for simplicity – as our next example shows.

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];  // One per physical processor for easy
     Local<int> i, j, k ;
     Local<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
        t = 0 ;
        for (j = 0 ; j < n ; j++) {
            t += AL[i * n + j] * BL[j * n + k] ;
        }
        C[PID] = t ;
     }
     C.sync() ;
   }
}
```

Our PRAM extension 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 and may be passed as parameters accordingly. Virtual parallelism and supersteps can be handled within such procedures. By also 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 EREW and CRCW 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 $\Theta(1)$ [LS96, ACS89]; 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 [McC95]. Thus, the locality of matrix multiplication is $\Theta(n/p^{1/3})$. Finally, optimal BSP sorting has been shown to have locality $\Theta(\lg(n/p))$ [Goo96]; by contrast, in [GS96c], it is shown that bitonic sorting has locality $\Theta(1 + (\lg n/p)/\lg^2 p)$.

For a comparison, 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 [MR85] employing $O(n/\lg n)$ processes. Matrix multiplication employs a straightforward n-process CREW algorithm. For sorting we use the non-optimal bitonic sorting algorithm [JáJ92] – 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 [GS96c]. 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.

	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 on a 4 processor SGI Shared Memory Power Challenge. We note that the slowdown factor decreases as the problem size increases and this may be attributed to the C++ PRAM library overheads. Similar findings are observed on the distributed memory Cray T3D.

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

	1 Physical Proc		4 Physical Procs	
Matrix	Direct	PRAM	Direct	PRAM
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	7.51	0.103	2.32
512 x 512	3.005	66.10	0.785	17.20

Table 2: Matrix Multiplication on SGI Power Challenge (time in seconds)

	Physical Processors							
	1		4		16		32	
Matrix	Direct	PRAM	Direct	PRAM	Direct	PRAM	Direct	PRAM
64 x 64	0.07	0.73	0.02	0.22	0.003	0.08	0.002	0.06
128 x 128	0.56	4.20	0.14	1.25	0.04	0.45	0.02	0.28
256 x 256	4.49	24.96	1.12	6.84	0.29	2.27	0.15	1.28
512 x 512	36.61		8.99	47.08	2.25	13.92	1.14	7.58

Table 3: Matrix Multiplication on Cray T3D (time in seconds)

library destroys locality and this is reflected in the slowdown of the simulation (refer to tables 2 and 3 and figure 2). Performance on the Cray T3D is respectable for large problems – an asymptotic limit of a factor of 5 slowdown was observed for all the different numbers of processors tested. 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 5-20 slowdown factor of the simulation is by increasing the data granularity, as we do for the sorting problem.

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 4 and figure 2). 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

	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 4: Bitonic Sorting on SGI Power Challenge.

PRAM algorithm to implement a portable semi-automatic BSP sorting algorithm.

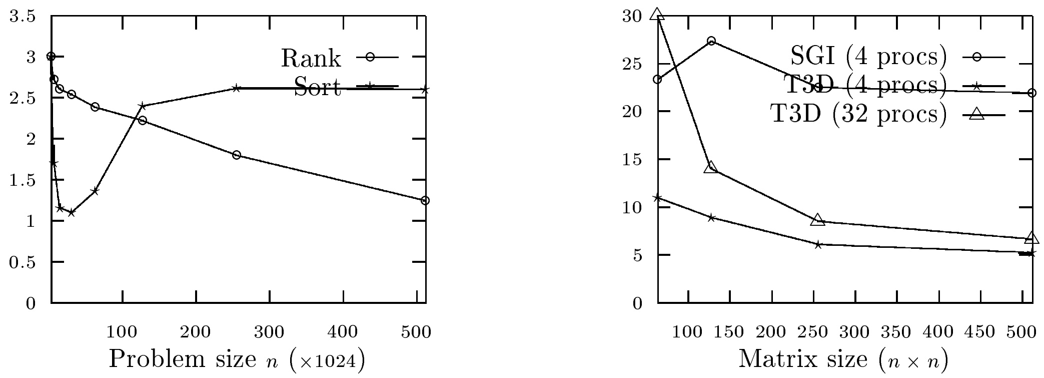

Figure 2: Slow-down results for list ranking, bitonic sorting and matrix multiplication.

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 very well on our system as compared to direct-mode implementations. Problems with overwhelmingly large locality (for example matrix multiplication) can also produce respectable performance – combining reduces the amount of message traffic generated from $2n^3/p$ requests per processor to at most $2n^2$ and at this level, the computation (n^3/p) dominates.

The PRAM is not to be confused with shared memory programming as provided by systems such as Treadmarks [LDCZ97]. The PRAM itself provides a discipline for memory access. For the CRCW model access is unrestricted although concurrent coincident writes will be arbitrarily resolved if they arise. For the EREW (exclusive read/write) memory accesses from each PRAM processor must never coincide. The vast collection of algorithms available in this disciplined stepwise synchronous model proves that this is not a hindrance. Techniques such as lock access and release [LDCZ97] are unnecessary in PRAM use. Another difference between the PRAM and shared memory programming is that algorithm design for the PRAM model typically assumes the number of processors to be massive - indeed for a problem of n elements typically we will expect n processors. In our approach performance prediction by retaining the simple predictable cost calculation of the PRAM and BSP models, and scalable performance are essential features. We expect virtual processor simulation and the lack of automatic caching, locking and block reads can lead to slow perfomance for some general programs. However our locality analysis and our results prove that for some well-known problems the approach is as efficient as systems such as direct BSP and the PVM, MPI or Treadmarks systems.

Our PRAM programming environment is easy to program, 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. Recent work verified that in practice the simulation is scalable to large numbers of physical processors (32, 64,..). 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 Alex Gerbessiotis and are indebted to Jon Hill his for valuable assistance with *BSPlib* throughout this work. We also thank an anonymous referee for helpful comments.

References

[ACS89] A. Aggarwal, A. Chandra, and M. Snir. On communication latency in PRAM computations. In *Proceedings of 1st ACM Symposium on Parallel Algorithms and Architectures SPAA 89*, pages 11–21, 1989.

[ACS90] A. Aggarwal, A. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, 1990.

[ADK+93] F. Abolhassan, R. Drefenstedt, J. Keller, W. Paul, and D. Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756–762, 1993.

[AGLS96] 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 SUPEUR'96 conference*, Krakow, Poland, September 1996.

[CF96] G. Cheng and G. C. Fox. Integrating multiple parallel programming paradigms in a dataflow-based software environment. *Concurrency: Practice and Experience*, 8(10):799–812, 1996.

[CFT+94] Lyndon J. Clarke, Robert A. Fletcher, Shari M. Trewin, R. Alasdair, A. Bruce, A. Gordon Smith, and Simon R. Chapple. Reuse, portability and parallel libraries. Technical Report tr9413, Edinburgh Parallel Computing Centre, The University of Edinburgh, 94.

[CM96] Z. J. Czech and W. Mikanik. Randomized PRAM simulation using T9000 transputers. In *Proceedings of HPCN 96*, number 678 in LNCS. Springer-Verlag, 1996.

[CMS95] A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Shared memory simulations with triple-logarithmic delay. *Lecture Notes in Computer Science*, 979:46–??, 1995.

[DM90] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of 17th ICALP*, number 443 in LNCS. Springer-Verlag, 1990.

[FKW96] A. Formella, J. Keller, and T. Walle. HPP: A high performance PRAM. *Lecture Notes in Computer Science*, 1124:425–??, 1996.

[Fox95] G. C. Fox. Software and hardware requirements for some applications of parallel computing to industrial problems. Technical Report SCCS-717, North East Parallel Applications Center, Syracuse University, 1995.

[FW78] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of 10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[GC92] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[Ger93] A. V. Gerbessiotis. *Topics in parallel and distributed computing*. PhD thesis, Harvard University, 1993.

[GHL+96] M. W. Goudreau, J. M. D. Hill, K. Lang, W. F. McColl, S. B. Rao, D. C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for the BSP worldwide standard library (preliminary version). Technical report, Oxford University Computing Laboratory, April 1996.

[Goo96] Michael T. Goodrich. Communication-efficient parallel sorting. In *Proc. of the 28th ACM Symp. on Theory of Computing (STOC)*, 1996.

[GS96a] 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*, Lyons, France, August 1996. Springer-Verlag.

[GS96b] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of 8^{th} ACM SPAA*. ACM Press, June 1996.

[GS96c] A. V. Gerbessiotis and C. J. Siniolakis. Primitive operations on the BSP model. Technical Report PRG-TR-23-96, Oxford University Computing Laboratory, October 1996.

[GV94] A. V. Gerbessiotis and L. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

[Har94] T. J. Harris. A survey of PRAM simulation techniques. *ACM Computing Surveys*, 26(2):187–206, June 1994.

[HSS94] T. Hagerup, A. Schmitt, and H. Seidl. FORK - a high-level language for PRAMs. Technical report, Universität des Saarlandes, April 1994.

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[KLM92] R. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the 24th STOC*, pages 318–326, 1992.

[KR90] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 870–941. Elsevier Science Publications, 1990.

[KT96] Christoph W. Keßler and Jesper Larsson Träff. A library of basic PRAM algorithms and its implementation in FORK. In *8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '96)*, pages 193–195, New York, USA, June 1996. ACM.

[LDCZ97] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Quantifying the performance differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computing*, 43(2):65–78, 15 June 1997.

[LMR95] Z. Li, P. H. Mills, and J. H. Reif. Models and resource metrics for parallel and distributed computation. In *Proceedings of 28th Hawaii International Conference on System Sciences (HICSS-28)*. IEEE, 1995.

[LRGD97] C. Leon, C. Rodrigeuz, F. Garcia, and F. De Sande. A PRAM oriented programming system. *Concurrency: Practice and Experience*, 9(3):163–179, 1997.

[LS96] D. S. Lecomber and K. R. Sujithan. Transgressing the boundaries: Unified scalable parallel programming. Technical Report TR-20-96, Oxford University Computing Laboratory, October 1996.

[McC93] W F McColl. General purpose parallel computing. In A Gibbons and P Spirakis, editors, *Lectures on Parallel Computation*, volume 4 of

Cambridge International Series on Parallel Computation, pages 337–391. Cambridge University Press, 1993.

[McC95] W. F. McColl. Scalable computing. *Lecture Notes in Computer Science*, 1000:46–, 1995.

[Mil93] R. Miller. A library for bulk-synchronous parallel programming. In *Proceedings of BCS Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing*, December 1993.

[MR85] Gary L. Miller and John H. Reif. Parallel tree contraction and its applications. In *26th FOCS*, pages 478–489, 1985.

[MV84] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[Ran93] A. Ranade. A framework for analyzing locality and portability issues in parallel computing. In F Meyer auf der Heide et al., editors, *Parallel architectures and their efficient use : First Heinz Nixdorf Symposium*, number 678 in LNCS. Springer-Verlag, 1993.

[RBJ88] A. Ranade, S. Bhatt, and S. Johnson. The fluent abstract machine. In *Proceedings of 5th MIT Conference on Advanced Research in VLSI*, pages 71–93. MIT Press, 1988.

[RPL96] Joy Reed, Kevin Parrott, and Tim Lanfear. Portability, predictability and performance for parallel computing: BSP in practice. *Concurrency: Practice and Experience*, 8(10):799–812, 1996.

[Sin96] C.J. Siniolakis. Direct bulk-synchronous parallel algorithms in computational geometry. Technical Report PRG-TR-10-96, Oxford Unversity Computing Laboratory, May 1996.

[SOHL+95] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, 1995.

[Suj96] K. R. Sujithan. Towards a scalable parallel object database - the bulk-synchronous parallel approach. Technical Report PRG-TR-17-96, Oxford Unversity Computing Laboratory, August 1996.

[Val90] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.