Bandwidth, Space and Computation Efficient PRAM Programming: The BSP Approach*

Alexandros G. Alexandrakis, Alexandros V. Gerbessiotis, David S. Lecomber, and Constantinos J. Siniolakis

Computing Laboratory, Oxford University, Wolfson Building, Oxford OX1 3QD, UK.
email : {c95aa,alexg,dsl,cs}@comlab.ox.ac.uk

Abstract

In this paper we investigate the tractability of PRAM simulations on the IBM SP2 system through the Bulk-Synchronous Parallel model. We present a portable C++ class library that provides PRAM style shared memory facilities for any parallel or distributed system. We also obtain almost optimal speedup for representative PRAM algorithms, thus demonstrating the suitability of our methods for high performance parallel systems.

1 Introduction

The Parallel Random Access Machine (PRAM) [1] has been one of the most widely used models of parallel computing. 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. The PRAM thus abstracts parallelism by stripping away considerations such as communication latency, memory and network conflicts during routing, bandwidth of interconnection networks, interprocessor communication, memory management, and processor synchronization. For this reason, algorithm design is simplified, and, as a consequence, numerous fine-grained PRAM algorithms (where the ratio of memory to processors is typically $O(1)$) have been proposed in the past several years, and a number of powerful techniques for designing such algorithms have been identified. Practical parallel algorithms however must be *scalable*, that is, applicable and efficient for an ample range of ratios of memory to processors. As noted in [5], most existing parallel computers follow the distributed memory model, where each processor has considerable local memory and computing power, and is connected to some interconnection network. For such machines, the ratio of memory to processors is substantially larger than $O(1)$.

Fine grained solutions with linear speedup rarely occur in real-life applications. This is mainly attributed to the fact that communication issues are important in parallel computing, and it is very likely that they will continue to be so as the number of processors per parallel system increases, despite advances in communication technology. The introduction of the Bulk-Synchronous Parallel (BSP) model of computation by L.G. Valiant [3] comes to address these limitations of parallel computing, by dealing explicitly with the notion of communication and synchronization among processors. The model has also been addressed in [4, 2, 5]

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

as a bridge between software and hardware, i.e. architecture independent software capable of delivering scalable performance for a wide class of applications on a variety of parallel computers [3].

2 The BSP Model

The *Bulk-Synchronous Parallel (BSP)* model of computation has recently been proposed [3, 4] as a unified framework for the design, analysis, and programming of general purpose parallel computing systems. The term bulk-synchronous reflects the underlying position of the model between the two extremes: entirely synchronous systems, and fully asynchronous systems. The BSP computer as described in [2, 5, 3, 4] consists of a set of processor-memory pairs, a global communications network, and a mechanism for the efficient barrier synchronization of all, or a subset, of the processors. The performance of any BSP computer can be precisely characterized by the following parameters.

(1) p - the number of processors.

(2) g - the ratio of the total throughput of the whole system in terms of basic computational operations, to the throughput of the communication network in terms of words of information delivered.

(3) 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*; a superstep may be conceived as a segment of a computation during which each processing unit performs a given thread using data already available there locally before the start of the superstep. The thread may include computation steps on values held locally at the start of the superstep, message transmissions, and message receipts. At the end of each superstep, all processing units wait in a global barrier synchronization until the non-local data exchanges are completed before proceeding to the next superstep. The time complexity of a superstep S in a BSP algorithm is determined as follows. Each superstep is charged $\max\{L, x + gh\}$ basic time steps, where x is the maximum number of basic computational operations executed by any processor during S, and h is the maximum number of messages transmitted or received by any processor.

Two modes of programming were envisaged on the BSP model:

(1) *automatic mode* where programs are written in a high level language that hides low level details such as memory allocation, and

(2) *direct mode* where the programmer retains control of memory distribution.

Direct mode programming has been addressed elsewhere [8, 2, 9, 10], and optimal results have been obtained for a multiplicity of problems, i.e. sorting, selection, string searching, geometric algorithms, data structures, linear algebra and numerical analysis. As noted in [2], 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 slack is required) than in automatic mode.

(3) the underlying BSP machine has high g (in contrast, automatic mode requires g to be constant for efficiency), and

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

Direct mode programming thus emphasizes 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.

3 Communication Efficient PRAM Simulations

In this section we discuss the practical implications of PRAM simulations on the BSP model. Given sufficient slack, we can design simulations that are one-optimal in communication operations and optimal to within small multiplicative constant factors in computational operations. By a PRAM simulation being one-optimal in communication we mean that $T(n)$ read and write operations of a PRAM can be simulated in $2(1+o(1))gT(n)/p$ time units on the BSP. The factor of $2g$ 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. Asymptotics are in terms of the problem size n as $n \to \infty$. The parameters g and p may or may not depend on n.

Problems for which the direct mode may possibly provide no better performance than automatic mode are problems with poor locality such as symbolic problems like list-ranking and sparse matrix computations. For such problems it is sufficient to employ automatic style BSP programming rather than direct style. A facility to simulate such PRAM algorithms on the BSP is thus desirable.

Simulations of CRCW and EREW PRAMs (CPRAMs and EPRAMs respectively) on the BSP model have been presented in [2]. It has been shown that the automatic mode achieves optimality within constant factors by simulating PRAM algorithms. The following is shown in [2].

Theorem 3.1 *Let ω_p be any function of p such that $\omega_p \to \infty$ as $p \to \infty$. Then the following amounts of slack are sufficient for simulating any one step of the EPRAM or CPRAM on the BSP model in one-optimal expected 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,

where the expressions before the hyphens denote the numbers of processors.

In both simulations one randomly chooses an appropriate hash function h that randomizes memory and distributes read and write requests evenly among the p processing units of the simulating BSP machine. The EPRAM and CPRAM simulations are outlined in figures 1 and 2 respectively. The simulations presented in these figures are necessarily oversimplifications. The actual simulations will do more than just those primitive operations. The major challenge lies in formulating a hashing strategy that will ensure that the memory requests of the simulated PRAM machine will be evenly distributed among the simulating processors.

We note that for simulations such as the pre-mentioned, where the PRAM processors are shared evenly among the processors of the BSP machine, and the memory space is randomized by hashing, parts (1) and (2) of theorem 3.1 are optimal in the sense that the removal of ω_p in their statement would render them false; this follows from [2].

```
begin EPRAM_simulation
   for all processors of the simulating BSP machine, in parallel
      do hash memory space ;
         distribute ω_p log p tasks of the simulated EPRAM to each
            processor of the simulating BSP machine ;
   for each processor of the simulating BSP machine, in parallel
      do perform the previously received ω_p log p tasks ;
end EPRAM_simulation
```

Figure 1: EPRAM simulation, p - number of processors.

```
begin CPRAM_simulation
   for all processors of the simulating BSP machine, in parallel
      do hash memory space ;
         distribute ω_p p log p tasks of the simulated EPRAM to each processor
            of the simulating BSP machine ;
   for each processor of the simulating BSP machine, in parallel
      do sort the up to ω_p p log p read (write) requests according to the
            processor destination using bucket sort ;
         hash the actual addresses in the destination procesor of each
            request to identify collisions (for each destination address
            one request is considered "unmarked" and the rest "marked") ;
         realize the "unmarked" requests ;
         communicate (in case of a read request) to the "marked" requests
            the values acquired by their "unmarked" representative ;
         perform the ω_p log p tasks ;
end CPRAM_simulation
```

Figure 2: CPRAM simulation, p - number of processors.

4 A Portable BSP C++ PRAM Class-Library

We describe here the implementation of PRAM style shared memory on a BSP machine; it is based on the methods presented in the preceeding sections. The implementation has been realized using the Oxford BSP library [6] which captures the BSP programming model in just six routines. The routines are callable from standard imperative languages, i.e. C++ and Fortran. Highly efficient, scalable, native implementations of the library are currently being developed at Oxford University Computing Laboratory for various parallel and distributed system architectures. The library for the IBM SP1 and SP2 distributed-memory systems uses MPL, the native message-passing library which is part of the IBM parallel operating environment.

The PRAM simulator provides a method of programming within the manageable framework of a BSP global superstep discipline. The package is fully dynamic and also generic - based on the C++ template style of programming. This allows a shared memory style array of any type (i.e. floats, integers or complex user defined structures) to be implemented. The shared memory class provides two constructs for accessing the shared memory - *read* and *write*. These two functions create requests that are then directed and handled transparently by the underlying simulator. The simulator guarantees that the accesses occur according to the BSP superstep principle.

The current implementation uses linear hash functions to implement the randomization of shared memory. Despite the theoretical limitations of such hash functions, experience on the IBM SP2 distributed memory system reinforces the general perception that in practice they are sufficient to deliver scalable performance. Such a hashing scheme has the added advantage of allowing efficient memory distribution.

We note that the PRAM simulator has been designed, analyzed, and implemented by employing the BSP cost model introduced in section 2.

5 EREW and CRCW Examples

Consider a linked list L of n nodes whose order is implicitly specified by an array A such that $A[i]$ contains a pointer to the node following node i on L, $i \in \{0, \ldots, n-1\}$. To simplify the exposition, we assume that $A[i] = 0$ when i is the end of the list. The *list-ranking problem* is to determine the distance of each node from the end of the list.

We study this problem because it is known to have poor locality and consequently does not benefit from direct-style BSP programming. Several PRAM algorithms have been proposed for this non-trivial problem. Vishkin and Cole [7] introduced an optimal technique for PRAM list-ranking and we implement this algorithm here. In table 1 we outline the results on an 8-node IBM SP2 system. In figure 3 we sketch the speedup of this implementation.

List Ranking			$M \times M$		
Problem Size	*Processors*	*Speedup*	*Dimensions*	*Processors*	*Speedup*
30000	1	1.00	50 × 50	1	1.00
	2	1.92		2	1.91
	4	3.63		4	3.61
	8	6.78		8	6.87
60000	1	1.00	75 × 75	1	1.00
	2	1.91		2	1.92
	4	3.63		4	3.63
	8	6.94		8	6.86
120000	1	1.00	100 ×100	1	1.00
	2	1.92		2	1.93
	4	3.65		4	3.67
	8	7.20		8	6.88

Table 1: List ranking and matrix multiplication on an 8-processor IBM SP2.

The second of our examples is matrix-matrix multiplication. This problem does benefit from direct mode programming, however we choose it as a simple demonstration of CREW programming with our PRAM simulator. In table 1 we outline the results on an 8-node IBM SP2. In figure 3 we sketch the speedup of this implementation.

6 Conclusion

In this paper we investigated the feasibilty of automatic mode BSP programming, where for the sake of simplicity the programmer is relieved of memory distribution. We have developed a BSP C++ PRAM portable class library to implement shared memory on the SP2 and obtained almost optimal speedup thus demonstrating the suitability of our methods for high performance parallel systems. In the future we intend to address higher order hash functions

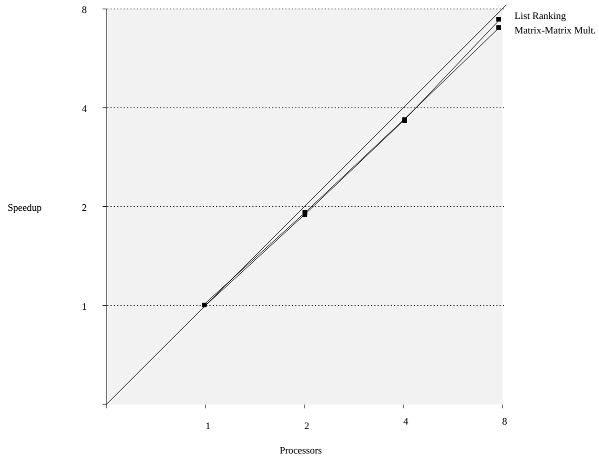

Figure 3: Speedup of EREW List Ranking and CREW Matrix Multiplication on 8-node IBM SP2.

and simulations with smaller slackness. We also intend to implement a larger pool of PRAM algorithms and assess their performance as compared to direct BSP algorithms.

References

[1] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the 10-th Annual ACM Symposium on Theory of Computing,* pp. 114-118, 1978.

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

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

[4] L. G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science,* (van Leeuwen, J., ed.), North Holland, 1990.

[5] W. F. McColl. BSP programming. In *Proceedings of DIMACS Workshop on Specification of Parallel Algorithms,* (G. E. Blelloch, K. M. Chandy, and S. Jagannathan, eds.), pp. 21-35, American Mathematical Society, May 1994.

[6] 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.

[7] R. Cole and U. Vishkin. Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking. *Information and Computation,* 70(1):32-52, 1986.

[8] C. J. Siniolakis. Bulk-Synchronous Parallel Algorithms in Computational Geometry. Technical Report PRG-TR-10-96, Oxford University Computing Laboratory, May 1996.

[9] 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,* Italy, to appear 1996.

[10] 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,* France, to appear 1996.