Portable and Scalable Algorithm Design on the IBM SP2: The Bulk-Synchronous Parallel Paradigm*

Alexandros G. Alexandrakis, Alexandros V. Gerbessiotis, and Constantinos J. Siniolakis

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

Abstract

The main reason for the moderate success of parallel computing has been the lack of a bridging and unifying model for parallel computation the way the von Neumann model has been for sequential computing. The introduction of the Bulk-Synchronous Parallel (BSP) model of computation by L.G. Valiant comes to address these limitations of parallel computing. In this paper we present our experiences in implementing BSP algorithms on an IBM SP2 system at the Oxford University Computing Laboratory. We designed and implemented several fundamental algorithms by using the Oxford BSP library. These algorithms include string searching, deterministic and randomized sorting, all-furthest-neighbor, three-dimensional convex hull, randomized selection, broadcast and parallel-prefix. We conclude that the IBM SP2 machine can be modeled as an ideal BSP computer capable of supporting scalable and portable algorithms that can be efficiently realized within the predicted performance.

1 Introduction

¿From the early days of sequential computing parallelism has been considered as a natural extension to sequential computing. The main reason for the current moderate success of parallel computing has been the lack of a bridging and unifying model for parallel computation the way the von Neumann model has been for sequential computing. One of the earliest attempts to model a parallel computer has been the Parallel Random Access Machine (PRAM). The PRAM model abstracts away considerations such as communication and synchronization. For this reason, algorithm design is simplified and, as a consequence, a considerable body of fine grained PRAM algorithms has been invented in the past, and a number of powerful techniques for designing such algorithms have been identified.

Nevertheless, since the idealized PRAM model fails to reflect constraints in the real world its practical importance is limited. In reality most problems do not as yet have fine grained algorithms with linear speedup, or such linear speedup algorithms are impossible due to bandwidth or diameter limitations. The Bulk-Synchronous Parallel (BSP) model of computation introduced by L.G. Valiant [1, 2] and subsequently treated in [3, 4, 5] addresses these limitations of parallel computing, by dealing explicitly with the notion of communication and synchronization among computational threads.

In this paper we present our experiences in implementing a wide range of BSP algorithms on an 8-processor IBM SP2 machine located at the Oxford University Computing Laboratory.

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

We designed and implemented several fundamental algorithms by using the Oxford BSP library [6]. A short list of algorithms that have been implemented includes:

(1) a randomized string searching algorithm *Ss* [7],

(2) a deterministic sorting algorithm *Ds* [8],

(3) two randomized sorting algorithms *Rs, Hs* [3, 9],

(4) an all-furthest-neighbor algorithm *Fn* [10],

(5) a three-dimensional convex hull algorithm *Ch* [10],

(6) a randomized selection algorithm *Rsel* [8],

(7) a broadcast and a parallel prefix algorithm *Broad, Prefix* [1, 3].

In all of our algorithms we used the BSP model in both the design and implementation stages of the algorithm development. That is, we based our design and implementation choices purely on the BSP cost model as described in the next section.

2 The BSP Model

The *Bulk-Synchronous Parallel* (BSP) model of computation has been proposed by L.G. Valiant [1, 2], as a unifying framework for the design, analysis and programming of all kinds of parallel and distributed systems, analogous to that which the von Neumann model has provided for sequential systems over the last fifty years. It offers the prospect of achieving both scalable parallel performance and architecture independent parallel software, and provides a framework that 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: entirely synchronous systems, and fully asynchronous systems. The BSP computer as described in [1, 2, 3, 4] consists of three parts: (a) a *collection of p processor/memory components* each consisting of a sequential processor with a block of local memory, (b) a *communication network* that can deliver messages point to point among the units, and (c) a *mechanism* for efficient *barrier synchronization* of all, or a subset, of the processing units. The parameters L and g are considered to be non-decreasing functions of the number of processors used by an algorithm.

The performance of any BSP computer can be characterized by the following three parameters: (a) p - the number of processing units, (b) 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; more formally, g is related to the time required to realize h-relations in continuous message traffic in gh time units [1], and (c) 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 processing unit is given a task to perform using data already available there locally before the start of the superstep. The task may include (a) computation steps on values held locally at the start of the superstep, (b) message transmissions, (c) message receipts. A superstep can complete at any time after the first L time units. The cost of a superstep S in a BSP algorithm is measured as follows. Each superstep is charged $\max\{L, x + gh\}$ time units, where x is the maximum number of local operations executed by any processor during S, and h is the maximum number of messages transmitted or received by any processor.

2.1 The Oxford BSP Library

Although, the BSP model, as previously described, may be regarded as an architectural model, it can be readily employed as a programming platform or, indeed, as a kind of programming paradigm. The underlying concept in the BSP model is the notion of the superstep, and the abstraction that non-local communication associated with a superstep takes place between supersteps as a global operation. Thus, a BSP program may be viewed as a succession of supersteps, with the required non-local communication occurring at the end of each superstep. Viewed in this way, the BSP model can be realized as a library of functions with architecture independent semantics for process creation, remote data access, and bulk synchronization. The Oxford BSP library [6] implements such a paradigm. The library functions are callable from standard imperative languages, such as C and Fortran. The Oxford BSP library was originally written on top of PVM. Currently, several versions of the library do exist that utilize TCP/IP, and the System V `shmat` call for multiprocessors systems with shared memory. For more details we refer to [6]. 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.

3 Fundamental Algorithms: Implementations and Experimental Results

We ran our experiments on an IBM SP2 system, consisting of eight IBM RS/6000 model processors connected by a high performance switch. All software has been written in C and was linked with the Oxford BSP library [6]. The source code was compiled using `cc` with option `-O3` under the `AIX` operating system. The BSP library function `bsp_time` was used for measuring running times [6]. For each of 3 test problem instances, we performed a series of experiments, and the running times indicated are the average of those experiments. The input consisted of uniformly distributed integers.

```
begin string_matching
   for each processor in parallel
      do compute hash-value h of pattern ;
         for each position i, i ∈ {1,...,n/p}, in the text
            do compute hash-value h_i of text ;
               if  h = h_i report possible match ;
end string_matching
```

Figure 1: Parallel randomized string matching, n - text size, p - number of processors.

The randomized string-searching algorithm *Ss* illustrated in figure 1 is based on ideas of [7] and has been refined to perform efficiently for large ratios of n/p. In figure 2 we consider a merge-based deterministic sorting algorithm *Ds* [8] that achieves close to linear speedup, by using a partitioning scheme (through regular sampling) which splits an arbitrary number of sorted sequences among the processors almost equally, independently of the input distribution. The partitioning scheme for each sorted sequence is found efficiently (using merging and parallel prefix on a small subset of the input) and only one merge pass is required. Similarly, in figure 3 we consider a randomized sorting algorithms *Rs* [3] that achieves linear speedup, by utilizing an efficient randomized partitioning scheme. We also present a variant of this randomized algorithm, denoted by *Hs* [9], that utilizes static data

```
begin deterministic_sorting
   for each processor in parallel
      do sort the input keys ;
   for all processors in parallel
      do select a regular sample S of some appropriate size ;
         sort S ;
         select S* of p-1 evenly distributed elements of S ;
         broadcast the previously selected elements ;
   for each processor in parallel
      do use S* to partition the input keys into p buckets ;
         route the keys of each bucket to the appropriate processor ;
         merge the previously received keys ;
end deterministic_sorting
```

Figure 2: Parallel deterministic sorting, p - number of processors.

```
begin randomized_sorting
   for all processors in parallel
      do select a random sample S of some appropriate size ;
         sort S ;
         select S* of p-1 evenly distributed elements of S ;
         broadcast the previously selected elements ;
   for each processor in parallel
      do use S* to partition the input keys into p buckets ;
         route the keys of each bucket to the appropriate processor ;
         sort the previously received keys ;
end randomized_sorting
```

Figure 3: Parallel randomized sorting, p - number of processors.

structures [11] in order to implement efficiently the partitioning scheme. The latter version of the randomized algorithm is expected to out-perform the former for small ratios of n/p.

The output-sensitive all-furthest-neighbor algorithm *Fn* is based on the work of [10]. In figure 4 we present a high-probability output-sensitive BSP algorithm *Ch* for computing the convex hull of points in three-dimensional space [10]. The algorithm is expected to perform particularly well when the points are drawn independently from any identically distributed continuous random function. The algorithm, however, seems to perform particularly well on both random and real data. Furthermore, the algorithm may be adapted to higher dimensions with minimal modifications. Finally, the randomized selection algorithm *Rsel* [8] considered in figure 5 is based on random sampling techniques and improves upon the multiplicative constant factors of all previously known results.

Table 1 summarizes the speedup observed for various problem and machine sizes. We note that the three-dimensional convex hull speedup has been normalized to account for the quadratic behavior of the underlying sequential algorithm. Several of our results exhibit super-linear speedup. There is a multiplicity of possible explanations for this effect, such as memory management and caching.

The broadcast and parallel prefix routines operate on trees of various depths [1, 3]. The ordinary binary tree implementation is the tree of depth 3 of table 2, for an 8-processor IBM SP2 machine. Block size is the size of a vector of integers employed either for broadcasting or for the computation of independent parallel prefix operations on the same associative

```
begin 3D_convex_hull
   set all processors to active ;
   while the number of active processors is greater than one ;
      for each active processor p_i, i ∈ {1,...,p-1}, in parallel
         do locally construct the convex hull CH(S_i) of the input points S_i ;
            discard all non-convex hull points ;
            concentrate remaining points into lower-indexed processors ;
            deactivate higher-indexed processors ;
   locally construct the convex hull of the remaining points on processor p_0 ;
end 3D_convex_hull
```

Figure 4: Parallel randomized three-dimensional convex hull, p - number of processors.

```
begin randomized_selection
   for all processors in parallel
      do select a random sample S of some appropriate size ;
         sort S ;
         select two appropriately chosen splitters s_1 and s_2 ;
         broadcast the previously selected splitters ;
   for each processor in parallel
      do use s_1 and s_2 to determine the elements S* between these splitters ;
         sort S* ;
         determine the required statistic ;
end randomized_selection
```

Figure 5: Parallel randomized selection, p - number of processors.

operator (addition).

We claim that the IBM SP2 machine can be abstracted as a BSP computer capable of supporting scalable and portable algorithms that can be efficiently realized without significant and unpredictable deterioration of performance. Our experimental results support and reinforce this statement.

References

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

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

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

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

[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.

		Ss	*Ds*	*Rs*	*Hs*	*Fn*	*Ch*	*Rsel*
Problem size	*Processors*	*Speedup*						
128000	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
(512000 for *Rsel*)	2	1.9	1.9	1.9	1.9	2.1	1.8	1.9
(800 for *Ch*)	4	3.4	3.3	4.0	3.7	3.6	2.8	3.6
	8	6.6	6.2	7.4	7.1	7.1	4.0	5.3
512000	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
(2048000 for *Rsel*)	2	1.9	1.8	1.9	2.1	2.2	1.8	2.0
(3200 for *Ch*)	4	3.9	3.4	3.7	4.0	4.2	3.5	3.8
	8	7.6	6.3	7.6	7.3	8.3	6.5	6.1
1024000	1	1.0	1.0	1.0	1.0	1.0	1.0	1.0
(4096000 for *Rsel*)	2	2.0	1.8	1.9	1.9	2.0	1.9	2.0
(6400 for *Ch*)	4	3.9	3.4	3.9	3.6	4.1	3.8	3.7
	8	7.3	6.4	7.5	6.5	8.2	7.4	6.8

Table 1: Experimental results on an 8-processor IBM SP2.

		Broad	*Prefix*
Block Size	*Tree Depth*	*Normalized Time*	
1	1	1.0	1.0
	2	1.2	1.1
	3	1.5	1.7
64	1	1.4	1.3
	2	1.3	1.5
	3	1.6	1.8
256	1	1.7	1.8
	2	1.6	1.5
	3	1.7	1.9

Table 2: Broadcasting and Parallel Prefix on an 8-processor IBM SP2.

[6] R. Miller. A library for Bulk-Synchronous Parallel programming. In *Proceedings of the British Computer Society Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, December 1993.

[7] R. M. Karp and M. O. Rabin. Efficient randomized pattern matching algorithms. *IBM Journal of Research and Development,* 31(2):249-260, 1987.

[8] A. V. Gerbessiotis C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP Model. In *Proceedings of the 8-th ACM Symposium on Parallel Algorithms and Architectures,* Italy, to appear 1996.

[9] A. V. Gerbessiotis and C. J. Siniolakis. Communication Efficient Data Structures on the BSP model with Applications. Technical Report, Computing Laboratory, Oxford University, 1996.

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

[11] A. V. Gerbessiotis 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.