The Bulk-Synchronous Parallel Random Access Machine

Alexandre Tiskin *

Oxford University Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD
United Kingdom
email: tiskin@comlab.ox.ac.uk

Abstract. The model of bulk-synchronous parallel (BSP) computation is intended to provide a simple and realistic framework for general-purpose parallel computing. Originally, BSP was defined as a distributed memory model. In this paper we present a new model, called BSPRAM, which is a variant of BSP based on a mixture of shared and distributed memory. The two models are equivalent for some important classes of algorithms. We identify two such classes: oblivious and coarse-block algorithms. Finally, we present BSPRAM algorithms for dense matrix multiplication and Fast Fourier Transform.

1 Introduction

The model of bulk-synchronous parallel (BSP) computation (see [14, 8, 10]) is intended to provide a simple and realistic framework for general-purpose parallel computing. Its main goal is to support the development of architecture-independent and scalable parallel software.

Previously many communication complexity models were proposed for parallel computing. One of the main divisions between the models is in the type of memory organisation: distributed or shared. Models based on shared memory provide the benefit of a single address space with uniform access. However, the cost of supporting shared memory in hardware is much larger than that of distributed memory.

Originally, BSP was defined as a distributed memory model with point-to-point communication between the processors. In this paper we present a variant of BSP based on a mixture of shared and distributed memory. This variant, called BSPRAM, is as simple and realistic as BSP. Its cost model is somewhat different from the BSP cost model; however, in some important cases the two models are equivalent.

Following [15], we say that a model A can *optimally simulate* a model B if there is a compilation algorithm that transforms any program with cost $T(n)$

*This work was supported in part by ESPRIT Basic Research Project 9072 — GEPPCOM (Foundations of General Purpose Parallel Computing).

on B to a program with cost $O(T(n))$ on A. If the compilation algorithm is defined only for programs which implement algorithms from some particular class, we say that A can optimally simulate B for that class of algorithms. We say that two models are *equivalent* (for a particular class of algorithms) if they can optimally simulate one another (for that class).

We identify two classes of algorithms (called oblivious and coarse-block algorithms, respectively), for which the models BSPRAM and BSP are equivalent. Algorithms from these two classes occur frequently in scientific computing. We give an example of such algorithm and its BSPRAM implementation, which is an adaptation of the McColl–Valiant BSP matrix multiplication algorithm.

2 The BSP model

A *BSP computer*, introduced in [13, 15, 14], consists of p *processors* connected by a communication network. Each processor has a fast *local memory*. The processors may follow different threads of computation. A BSP computation is a sequence of *supersteps*. A superstep consists of an *input phase*, a *computation phase* and an *output phase*. In the input phase a processor receives data that were sent to it in the previous superstep; in the computation phase it performs local computations; in the output phase it can send data to be received by other processors in the next superstep. The processors are synchronised between supersteps; the computation within a superstep is completely asynchronous.

The *cost unit* is the cost of performing a basic arithmetic operation or a local memory access. If for a particular superstep w is the maximum number of local operations performed by each processor, h' (respectively h'') is the maximum number of data units received (respectively sent) by each processor, and $h = h' + h''$, then the cost of the superstep is defined as $w + h \cdot g + l$. Here g and l are parameters of the computer. The value g is called *communication throughput ratio* (also sometimes *bandwidth inefficiency* or *gap*), the value l — *communication latency* (also sometimes *synchronisation periodicity*). We write BSP (p, g, l) for an instance of BSP with the given values of p, g and l. The values of w and h typically depend on the number of processors p and on the problem size. If a computation consists of S supersteps with costs $w_r + h_r \cdot g + l$, $1 \leq r \leq S$, then its total cost is $W + H \cdot g + S \cdot l$, where $W = \sum_r w_r$, $H = \sum_r h_r$.

The BSP model does not support special broadcasting or combining facilities. The papers [15, 14] address the issue of simulating a PRAM on a BSP computer with constant throughput ratio. Since the considered version of PRAM is CRCW, such simulation provides a mechanism for broadcasting and combining. In [5, 6] the BSP model augmented by explicit broadcasting and combining mechanisms is suggested; however, arbitrary broadcasting and combining of messages at constant cost are dismissed in [6] as unrealistic. In the following sections we present a new approach to broadcasting and combining in BSP and show that in some important cases it can be efficiently implemented in standard BSP.

Being a general-purpose computation cost model, BSP is also intended to serve as a basis for a simple and efficient programming model. PRAM may be

used as such a model via the simulation from [15, 5]; however, this approach is efficient only for small values of g, and even in these cases is rather complex. Generally, in order to utilise efficiently the computer resources a typical BSP program should consider the values p, g and l as configuration parameters. Therefore, the goal is to design parametrised parallel software. For most problems, a balanced distribution of data and computation work will lead to algorithms that simultaneously achieve optimal computation, communication and synchronisation costs; however, for some other problems a need to trade off these costs will arise. For example, broadcasting of a single value from a processor exhibits a communication-synchronisation tradeoff: it can be performed with $H = S = O(\log p)$ by a balanced binary tree, or with $H = O(p)$ and $S = O(1)$ by sending the value directly to every processor. However, if p values are to be broadcast from one processor, the tradeoff disappears: by scattering the values so that each one resides in a separate processor, and then performing total exchange, the problem can be solved with $H = O(p)$ and $S = O(1)$, which is obviously optimal. The domain of matrix computation provides further examples of both kinds of problems: for instance, matrix multiplication can be done optimally in communication and synchronisation, but matrix inversion exhibits a communication-synchronisation tradeoff with a polynomial range of parameters.

The BSP model is successful in capturing the important features of a large number of existing parallel systems. This primarily due to its generality and simplicity. However, in certain situations the BSP model still appears to be unnecessarily prescriptive. One such area is subset synchronisation of processors. If BSP is used purely as an abstract algorithm development platform, the need to synchronise some of the processors, rather than all of them, does not arise frequently. On the other hand, the ability to set aside a subset of processors to perform a particular independent task is desirable, and sometimes even essential, for a programming model based on BSP. The need for subset synchronisation was acknowledged in [14] but no sufficiently general cost model for BSP subset synchronisation has been proposed so far.

Another feature lacking in standard BSP is an input-output model. This forces an algorithm designer to make assumptions on input and output that are not inherent to the nature of the problem solved. For example, the BSP matrix multiplication algorithm from [10] requires that the multiplied matrices are distributed evenly across the processors' local memories, and forms the product distributed in the same way. Effectively, the input and output data distribution forms a part of the problem, and does not necessarily reflect the actual algorithmic context in which the problem is being solved. Results on PRAM simulation are intended to overcome this difficulty; however, efficient PRAM simulation is possible only when the communication throughput is constant.

In the following sections of this chapter we propose a different solution: to introduce elements of shared memory architecture into BSP, while retaining the bulk-synchronous computation structure and the parameters g and l. This is done in two steps: first, we define an intermediate model BSP+, in which messages can stay in the network for more than one superstep, so the network may

be regarded as single-write, single-read memory for the messages. After that, we consider a BSP-type model in which the network is implemented as a conventional random-access shared memory unit. We call the new model BSPRAM. It is as simple and realistic as BSP, and we will identify some important cases when it is equivalent to BSP. At the same time the BSPRAM model is more convenient for algorithm design and programming than plain BSP.

3 A modification of BSP

This section describes the first step in augmenting BSP by random-access shared memory.

We start with the variant of standard BSP in which non-local data access is expressed by explicit send and receive statements. Consider the following example. A value x, which is at the beginning of superstep 0 local to a processor 0, must be used at some point by each of the p processors. However, not all the processors need it at the same time. Processor 1 starts computation with x at superstep 1, processor 2 at superstep 2, and so on until the processor $p-1$, which needs x starting from superstep $p-1$. Such dependency pattern arises, for example, when solving a triangular linear system by forward- or back-substitution, or a general linear system by Gaussian elimination (see e.g. [9, 10]). A simple broadcast in superstep 0 would solve the problem, as would the policy of sending x from processor 0 directly to processor q at superstep $q-1$. However, these are not necessarily the most efficient ways. For example, the paper [9] presents a systolic-type algorithm, which consists in sending x from processor $q-1$ to processor q at superstep $q-1$. Our goal is to obtain a generic method unifying all these different approaches.

Let us relax the requirement that corresponding send and receive statements must be issued in the adjacent supersteps. The computation still proceeds in asynchronous supersteps and synchronised between the supersteps; however, once sent, a message may stay in the network for indefinitely many supersteps before it is received. We call a message *slow* if its receipt is delayed by at least one superstep. The cost of the send and receive statements in the extended model is determined in the same way as in the standard BSP model. A message is counted in the cost when it sent or received; no cost is incurred by keeping a message in the network for as many supersteps as necessary.

We call the new model *BSP+*. It is more powerful than BSP, in the sense that it allows some computations to be performed at a lower cost than in BSP. However, the two models are equivalent for the following important class of algorithms. We call an algorithm *oblivious* if every processor executes the sequence of instructions that does not depend on the input (the data processed by these instructions may, of course, be different for different inputs). From this definition it follows that the size of the input to an oblivious algorithm must be fixed (otherwise it would be impossible to read the input always by an identical sequence of instructions). For example, any algorithm that can be represented by an arithmetic circuit is oblivious. For oblivious algorithms we have the following result.

Lemma 1. *The model BSP+ is equivalent to the standard BSP for the class of oblivious algorithms.*

Proof. Since the sequence of operations in an oblivious algorithm is known in advance, we only need to show that any computation on BSP+ can be performed on the standard BSP at the same asymptotic cost. Let us consider one superstep of such computation with communication cost $h = h' + h''$, where h' (respectively h'') is the cost of the input (respectively, output) phase. Partition the instructions of the input (respectively, output) phase into h' (respectively h'') sets, so that in each set all the instructions are performed by different processors. Represent the whole computation as a graph G in which the nodes correspond to the sets of input-output instructions, and the edges represent messages. Two nodes v_1 and v_2 are connected by an edge e if the message represented by e is sent by one of the instructions in the set represented by v_1 and received by one of the instructions in the set represented by v_2. The graph G is bipartite with maximum degree at most p. By a well-known property of bipartite graphs (see e.g. [2], p. 247), there is a colouring of the edges of such a graph with not more than p colours, where all the edges connected to one and the same vertex are coloured differently. The simulation is achieved by using the local memory of the processor corresponding to the color of an edge e as intermediate storage for the message represented by e. For each message, an extra receive and an extra send are needed; however, by the graph colouring construction, this increases the communication cost by not more than a factor of 2. Also, an extra superstep between every two supersteps of the original computation is necessary; this increases the synchronisation cost by not more than a factor of 2. The computation cost remains unchanged. ■

The use of slow messages can have a significant impact on algorithm performance. Some inter-processor links will inevitably be less efficient than the others, and therefore may be dedicated specifically to routing slow messages. From the programmer's point of view, however, BSP+ does not differ much from BSP, since they both support distributed memory programming with message passing primitives. To allow shared-memory style BSP programming, the BSP+ model needs further modification, which is carried out the next section.

4 The BSPRAM model

In the BSP+ model introduced in the previous section, the network acts not only as a message delivery mechanism, but also as a memory for keeping slow messages as long as necessary. The only significant aspect in which it differs from conventional random-access memory is that each message has an explicit destination address, whereas it is not generally known in advance which processor will read from a memory location. However, in some important cases it is still possible to organise computation so that when a value is being written, one can

determine which processor will read it, and at what superstep it will happen. This is always the case, for example, when the executed algorithm is oblivious.

The above observation justifies the introduction of a new model based on BSP, which in addition to the processors' local memories has a random-access shared memory unit replacing the point-to-point communication network. Apart from simplifying the programming, this new version of BSP provides a basic general purpose input-output model, in which the data are input from and output to the shared memory.

Thus, a two-level memory model of parallel computation is proposed. Since it is closely related both to BSP and to PRAM, we will call it *BSPRAM*. Like a BSP computer, a BSPRAM consists of p *processors* with fast *local memories*. In addition, there is a single shared *main memory*. As in BSP, the computation proceeds by *supersteps*. A superstep consists of an *input phase*, a *computation phase*, and an *output phase*. In the input phase a processor can read data from the main memory; in the computation phase it performs local computation; in the output phase it can write data to the main memory. The processors are synchronised between supersteps; the computation within a superstep is completely asynchronous.

The PRAM is usually considered in a number of versions. Among them are EREW PRAM, which requires that every location is read from or written to by not more than one processor in any step, and CRCW PRAM, which allows several processors to read from of write to a location concurrently in one step. Similarly, the definition of BSPRAM leaves a choice of allowing or disallowing concurrent access to the main memory in one superstep. We consider an *exclusive-read, exclusive-write BSPRAM (EREW BSPRAM)*, in which every location of the main memory can be read from and written to only once in every superstep, and a *concurrent-read, concurrent-write BSPRAM (CRCW BSPRAM)*, which has no restrictions on concurrent access to the main memory. For convenience of algorithm design we assume that if a value x is being written to a main memory location containing the value y, the result may be determined by any prescribed function of x and y, computable in constant time. Similarly, if values $x_1, \ldots, x_n$ are being written concurrently to a main memory location containing the value y, the result may be determined by any prescribed function of $x_1, \ldots, x_n, y$, computable in time linear in n. This corresponds to resolving concurrent writing in PRAM by combining (see e.g. [4]).

The cost of a BSPRAM superstep is defined, similarly to the BSP model, as $w + h \cdot g + l$. Here w is the maximum number of local operations performed by each processor, and $h = h' + h''$. The value of h' (respectively h'') is defined as the maximum number of data units read from (respectively written to) the main memory by each processor in the superstep. As in BSP, the values g and l are fixed parameters of the computer. We write BSPRAM (p, g, l) for an instance of BSPRAM with the given values of p, g and l. The cost of a computation consisting of several supersteps is defined as $W + H \cdot g + S \cdot l$, where W, H and S have the same meaning as in the BSP model.

It is clear that EREW BSPRAM can optimally simulate BSP. For oblivious algorithms the converse is also true.

Theorem 1. *BSP (p,g,l) can optimally simulate EREW BSPRAM (p,g,l) for the class of oblivious algorithms.*

Proof. Having established Lemma 1, we only need to show how BSP+ can optimally simulate an oblivious EREW BSPRAM algorithm.

Consider an EREW BSPRAM computation. First, we transform it so that every used location in the main memory is read from and written to exactly once. In order to do this, we replace every read instruction by a read-write instruction pair. The read instruction in the pair reads the value from the main memory; the write instruction, performed in the output phase, writes the value back the main memory to a previously unused location. We also replace every write instruction (which possibly involves combining of the value being written with the value already stored in the target location) by a read-write instruction pair. The read instruction in the pair is performed in the input phase; it reads the value previously stored in the target location, if any. The write instruction writes the output value, possibly combined with the old contents of the target location, to a previously unused location. There is no interference, since reading and writing within a superstep are exclusive. The transformation increases the communication cost by not more than a factor of 2, and leaves the synchronisation cost unchanged.

To achieve the BSP+ simulation of the transformed EREW BSPRAM computation, it only remains to replace all main memory access instructions by corresponding message passing ones. Namely, writing to a main memory location is simulated by sending a message (the destination address of which is known since the algorithm is oblivious), and subsequent reading from the same location is simulated by receiving the message. This part of the simulation leaves the communication and synchronisation costs unchanged. ■

In the following sections we shall give examples of using EREW BSPRAM for executing oblivious algorithms. We now consider a different class of structured algorithms, which occur even more frequently in scientific computation. We say that a set of locations in the main memory of BSPRAM constitutes a *block*, if in any given superstep any processor that reads from (respectively, writes to) one of the locations reads from (respectively, writes to) all of them. Informally, a block is treated as "one whole piece of data". For example, if the contents of n main memory locations of CRCW BSPRAM is broadcast to all the processors by concurrent reading from the locations, then the locations form a block of size n. The communication cost of such broadcast is $H = O(n)$.

We say that a BSPRAM computation is *coarse-block* if all the data in the main memory are organised in blocks of size not less than p. Thus, the broadcasting algorithm mentioned above is coarse-block if $n \geq p$. In the latter case it is possible to simulate the algorithm optimally in BSP by applying the "scatter and total-exchange" technique described in Section 2. The same principle lies behind the McColl–Valiant matrix multiplication algorithm from [10]. The idea can be generalised in the following way.

Theorem 2. *BSP (p, g, l) can optimally simulate CRCW BSPRAM (p, g, l) for the class of coarse-block algorithms.*

Proof. Choose an arbitrary balanced distribution of every main memory block across the processors. Reading from and writing to a block are simulated by corresponding send and receive instructions. The simulation increases the communication and synchronisation costs by not more than a factor of 2. ■

In comparison with BSP, the BSPRAM model allows an easier design and analysis of parallel algorithms. It may also prove to be more convenient for practical programming. In particular, the algorithm designer is no longer required to specify the distribution of input and output, since for most nontrivial problems it is natural to assume that they are kept in the main memory. For algorithms composed from several stages, each with its own input and output, the new model will help to ensure that the data distribution is consistent throughout the algorithm.

On some parallel computers, a direct implementation of the BSPRAM model may prove practical. In any case, the proofs of Theorems 1 and 2 show that a BSP computer can execute many important BSPRAM algorithms within a low constant factor of their BSPRAM cost. The next sections give two examples of such algorithms.

5 Matrix multiplication in BSPRAM

In this section we develop and analyse a BSPRAM algorithm for one of the most common problems in scientific computation: dense matrix multiplication. The cost analysis of the algorithm, as well as of most BSP algorithms in general, is carried out in the assumption that the input size n is sufficiently large with respect to p. Since the natural "degree of parallelism" of an algorithm is a function of n, the requirement that n should be large is essentialy the *parallel slackness* requirement, which is a necessary condition for simulation of a PRAM on BSP (see [14]). However, we exploit this slackness in a different way, obtaining a direct CRCW BSPRAM algorithm, rather than performing it on a simulated PRAM under the condition of constant g. The cost analysis is directly applicable to a computation in the BSP model as well, since the algorithm is coarse-block (for sufficiently large problem size).

The problem is to compute the matrix product $XY = Z$, where X, Y, Z are arbitrary $n \times n$ matrices in main memory. This problem is of great importance, and, somewhat surprisingly, significant theoretical complexity. Since the groundbreaking paper by Strassen [12] much work on the sequential cost of matrix multiplication has been done. However, no lower bound asymptotically better than trivial $\Omega(n^2)$ has been found; nor there is any indication that the current $O(n^{2.376})$ algorithm from [3] is close to optimal.

We aim at parallelising the standard $O(n^3)$ method without using fast matrix multiplication techniques. The method consists in a straightforward computa-

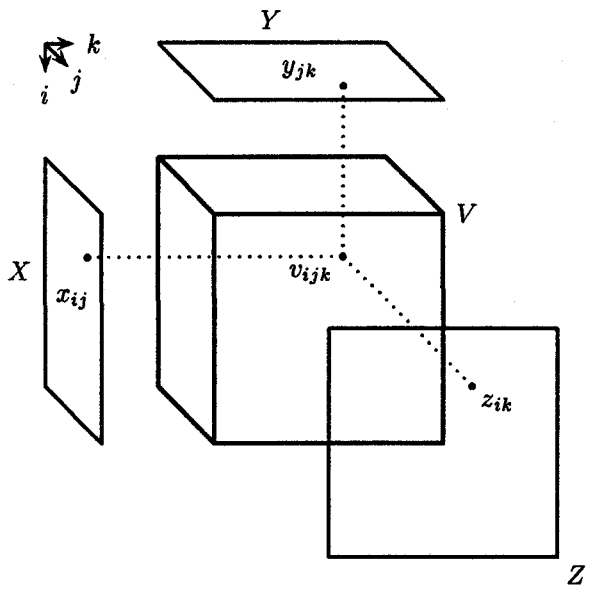

Figure 1: McColl–Valiant matrix multiplication: data dependencies.

tion of the family of bilinear forms

$$z_{ik} = \sum_{j=1}^{n} x_{ij} y_{jk} \qquad 1 \le i, k \le n \tag{1}$$

The algorithm is derived from the McColl–Valiant BSP algorithm for matrix multiplication described in [10], which in turn is based on an idea from [1]. Following (1), we need to compute the array of products $V = (v_{ijk})$, where

$$v_{ijk} = x_{ij} y_{jk} \qquad 1 \le i, j, k \le n$$

We represent the array V as a cube of volume n^3 in integer three-dimensional space (see Figure 1). Each of the arrays X, Y, Z is represented as the projection of this cube onto the coordinate planes $k = 0$, $i = 0$ and $j = 0$, respectively. The element v_{ijk} depends on its first two projections x_{ij} and y_{jk} and contributes to the computation of its third projection z_{ik}. It only remains to partition the cube in a straightforward way. We divide the cube V into a regular grid of p smaller cubic subarrays of volume n^3/p, and assign to each processor the problem of computing one of the subarrays.

Algorithm 1. *Multiplication of square matrices of size n (see Figure 1).*

Input: matrices $X = (x_{ij})$ and $Y = (y_{ij})$, $1 \le i, j \le n$.

Output: a matrix $Z = (z_{ij})$, $1 \le i, j \le n$, which is the product of X and Y.

Description. The CRCW BSPRAM computation proceeds in one superstep. Every projection of a subarray constitutes a block. A processor reads from the main memory the projections of a subarray along the axes k and i, computes the subarray elements and sums the elements in groups along the axis j. The result of local computation is the combined contribution of the subarray to its projection along the axis j; we call this contribution a *partial projection* of the subarray along the axis j. The processors write the computed partial projections concurrently to the main memory. Concurrent writing is resolved by addition of the values written; this combines the partial projections of the subarrays into the full projection of the array V. The resulting array Z is the matrix product of X and Y. The slackness required by Theorem 2 is $n \geq p^{5/6}$, but this can be easily improved to $n \geq p^{1/2}$.

Cost analysis. The computation cost W is clearly $O(n^3/p)$. The communication cost is proportional to the block size, i.e. the integer area of a block projection: $H = O(n^2/p^{2/3})$. The synchronisation cost is $S = O(1)$. It can be shown that these cost values are optimal for any BSPRAM parallelisation of the standard matrix multiplication method. The algorithm is coarse-block, therefore this cost analysis also applies to the BSP model by Theorem 2. The algorithm is also oblivious; however, Theorem 1 does not apply, since the CRCW version of BSPRAM is used. ■

Note that, unlike [10], no assumption on the distribution of input and output data was necessary. All the input and output data are assumed to reside in the main memory. The optimality of Algorithm 1 can be shown by an input-output complexity argument.

Algorithm 1 can serve as a building block for development of more advanced matrix algorithms, such as matrix inversion or linear system solvers.

6 Fast Fourier Transform in BSPRAM

Fast Fourier Transform is one of the most important algorithms in scientific computing (see e.g. [4, 7]). Its data dependency pattern, the butterfly dag, also arises in other algorithms — for example, it can be used for computing parallel prefix. As observed in [11, 14], the butterfly graph can be partitioned in a way perfectly suitable for bulk-synchronous parallel computation. Every level of the butterfly consists of $n/2$ independent tasks, each of size 2. Similarly, any k consecutive levels consist of $n/2^k$ independent tasks, each of size 2^k. Independent parallel tasks may be performed in one superstep on different processors. If the slackness is sufficiently large, two supersteps will suffice to complete the computation.

Figure 2 shows the two-superstep FFT algorithm in the case of $n = 16$. Each superstep consists of four independent tasks of size 4; therefore, the slackness is sufficient for the use of $p \leq 4$ processors. In the following general BSPRAM algorithm it is assumed that the input and output data reside in the main memory.

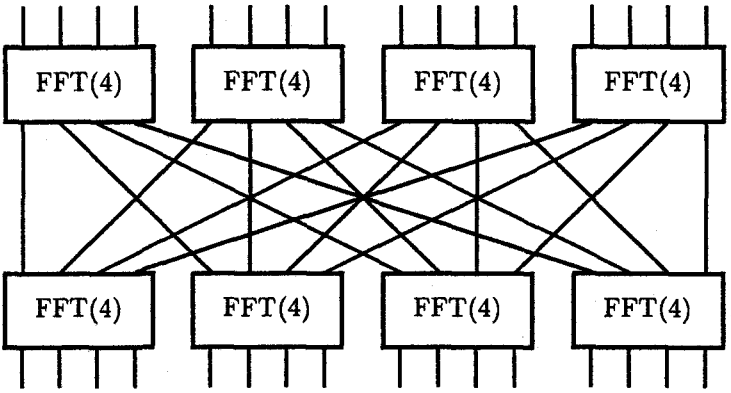

Figure 2: Fast Fourier Transform of 16 values.

Algorithm 2. *Fast Fourier Transform of n elements (see Figure 2 for $n = 16$).*

Input: an array $\mathbf{x} = (x_i)$, $0 \leq i < n$.

Output: the Fourier transform of the array $\mathbf{x}$.

Description. The $\log n$ levels of the n-input butterfly are computed on EREW BSPRAM (p, g, l). The computation proceeds in two supersteps, each comprising $1/2 \cdot \log n$ levels. Each of the supersteps is composed of $n^{1/2}$ independent FFT tasks of size $n^{1/2}$; therefore, each processor is assigned $n^{1/2}/p$ tasks. The required slackness is $n \geq p^2$.

Cost analysis. The computation cost W is clearly $O(n \log n/p)$. The communication cost is $H = O(n/p)$. The synchronisation cost S is constant. The values of W, H and S are trivially optimal. The algorithm is oblivious, therefore this cost analysis also applies in the BSP model by Theorem 1. The algorithm is coarse-block if the slackness is at least $n \geq p^3$. ■

The question of reducing the slackness required for BSP implementation of Fast Fourier Transform is addressed in [14].

7 Conclusions

A new model for bulk-synchronous parallel computing, the BSPRAM model, has been presented. It was shown that it in some important cases it is equivalent to the BSP model. BSPRAM contains elements of shared memory and therefore simplifies the design and analysis of bulk-synchronous parallel algorithms. We have given two examples of algorithm design in the BSPRAM model. In these examples, the BSPRAM algorithms for standard (non-fast) matrix multiplication and Fast Fourier Transform were derived from the corresponding optimal BSP algorithms. The BSPRAM approach has allowed to express the algorithms

in the shared-memory style, and to remove the assumption of even input and output distribution, which was necessary in their BSP versions.

References

1. A Aggarwal, A K Chandra, and M Snir. Communication complexity of PRAMs. *Theoretical Computer Science*, 71:3–28, 1990.
2. C Berge. *Graphs*, volume 6, part 1 of *North-Holland Mathematical Library*. North-Holland, second revised edition, 1985.
3. D Coppersmith and S Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.
4. T H Cormen, C E Leiserson, and R L Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. The MIT Press and McGraw–Hill, 1990.
5. A V Gerbessiotis and L G Valiant. Direct bulk-synchronous parallel algorithms. Technical Report TR–10–92 (Extended version), Aiken Computation Laboratory, Harvard University, 1992. Shorter version appears in Proc. 3rd Scandinavian Workshop on Algorithm Theory, July 8–10, 1992, LNCS Vol. 621, pp. 1–18, Springer-Verlag.
6. M Goodrich. Communication-efficient parallel sorting. In *Proceedings of the 28th ACM Symp. on Theory of Computing (May 1996)*.
7. J JáJá. *An Introduction to Parallel Algorithms*. Addison–Wesley, 1992.
8. 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*, chapter 13, pages 337–391. Cambridge University Press, 1993.
9. W F McColl. Special purpose parallel computing. In A Gibbons and P Spirakis, editors, *Lectures on parallel computation*, volume 4 of *Cambridge International Series on Parallel Computation*, chapter 13, pages 261–336. Cambridge University Press, 1993.
10. W F McColl. Scalable computing. In J van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 46–61. Springer-Verlag, 1995.
11. C H Papadimitriou and M Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the 20th Annual Symposium on Theory of Computing*, pages 510–513, 1988.
12. V Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.
13. L G Valiant. Bulk-synchronous parallel computers. In M Reeve, editor, *Parallel Processing and Artificial Intelligence*, chapter 2, pages 15–22. John Wiley & Sons, 1989.
14. L G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
15. L G Valiant. General purpose parallel architectures. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 18, pages 943–971. Elsevier, 1990.