ELSEVIER

Theoretical Computer Science 196 (1998) 109–130

Theoretical Computer Science

The bulk-synchronous parallel random access machine

Alexandre Tiskin*,[1]

Oxford University Computing Laboratory, Wolfson Building, Parks Rd, Oxford OX1 3QD, UK

Abstract

The model of bulk-synchronous parallel (BSP) computation is an emerging paradigm of general-purpose parallel computing. Originally, BSP was defined as a distributed memory model. Shared-memory style BSP programming had to be provided by PRAM simulation. However, this approach destroys data locality and therefore may prove inefficient for many practical problems. In this paper we present a new BSP-type model, called BSPRAM, which reconciles shared-memory style programming with efficient exploitation of data locality. BSPRAM can be optimally simulated by BSP for a broad range of algorithms. We identify some characteristic properties of such algorithms: obliviousness, slackness, granularity. Finally, we illustrate these concepts by presenting BSPRAM algorithms for butterfly dag computation, cube dag computation, dense matrix multiplication and sorting.

Keywords: BSP computing; Automatic memory management; PRAM simulation; Shared memory simulation; BSP algorithms

1. Introduction

The model of bulk-synchronous parallel (BSP) computation (see [27, 18–20]) is intended to provide a simple and practical framework for general-purpose parallel computing. Its main goal is to support the creation of architecture-independent and scalable parallel software. The key features of BSP are the treatment of the communication medium as an abstract fully connected network, and explicit and independent costing of communication and synchronisation.

Many other communication complexity models have been proposed for parallel computing. One of the main divisions among the models is by the type of memory organisation: distributed or shared. Models based on shared memory are appealing from

* E-mail: tiskin@comlab.ox.ac.uk.

[1] This work was supported in part by ESPRIT Basic Research Project 9072 – GEPPCOM (Foundations of General Purpose Parallel Computing).

0304-3975/98/$19.00
PII S0304-3975(97)00197-7

the theoretical point of view, because they provide the benefits of natural problem specification, convenient design and analysis of algorithms, and straightforward programming. For this reason, the PRAM model has dominated the theory of parallel computing. However, this model is far from being realistic, since the cost of supporting shared memory in hardware is much higher than that of distributed memory. Consequently, much effort was put into the development of efficient methods for simulation of PRAM on more realistic models.

Unlike PRAM, BSP accurately reflects main design features of most existing parallel computers. On the abstract level BSP is defined as a distributed memory model with point-to-point communication between the processors. Paper [28] shows how shared-memory style programming, with all the associated benefits, can be provided in BSP by PRAM simulation. However, this approach does not allow the algorithm designer to exploit data locality, and therefore in many cases may lead to inefficient algorithms. In this paper we propose a new model, called BSPRAM, which stands between BSP and PRAM. BSPRAM is based on a mixture of shared and distributed memory, and allows one to specify, design, analyse and program shared-memory style algorithms that exploit data locality. The cost models of BSPRAM and BSP are based on the same principles, but there are important differences connected with concurrent memory access in BSPRAM. The two models are related by efficient simulations for a broad range of algorithms.

We identify some properties of a BSPRAM algorithm that suffice for its optimal simulation in BSP. Algorithms possessing at least one of these properties – obliviousness, high slackness, high granularity – are abundant in scientific and industrial computing. We show the meaning and use of such properties on several examples: butterfly dag computation, cube dag computation, matrix multiplication, sorting. In view of our simulation results, BSPRAM here plays a role of a methodology for generic BSP algorithm design.

Algorithms presented in this paper, as well as many other BSPRAM algorithms, are defined for input sizes that are sufficiently large with respect to the number of processors. Apart from simplifying the algorithms, this condition provides slackness and granularity necessary for their efficient BSP simulation. A typical form of such condition is $n \geqslant poly(p)$, where n is the size of the input, p is the number of processors, and *poly* is a low-degree polynomial. Practical problems usually satisfy such conditions, unless the number of processors is extremely large. Because of that, we present the algorithms in their simplest form, without trying to adapt them for lower values of n. Instead, we only note where such optimisation is possible, and give references to papers that address this problem.

For the sake of simplicity, throughout the paper we ignore small irregularities that arise from imperfect matching of integer parameters. For example, when we write "divide an array of size n into p regular blocks", the value n may not be an exact multiple of p, and therefore the blocks may differ in size by ± 1. Such effects need not be considered in the abstract description of algorithms, since they can be easily accounted for during implementation.

2. Historical background

The last fifty years have seen a tremendous success of sequential computing. As pointed out in [27, 18, 19], this was primarily due to the existence of a single model, the von Neumann computer, which was simple and realistic enough to serve as a universal basis for sequential computing. No such basis existed for parallel computing. Instead, there was a broad variety of hardware designs and programming models.

One of the main traditional divisions among models of parallel programming is the organisation of memory: distributed versus shared. Shared memory is much costlier to support in hardware than distributed memory. However, shared memory has some important advantages:

- natural problem specification – computational problems have well-defined input and output, that are assumed to reside in the shared memory. As a contrast, algorithms for a distributed memory model have to assume a particular distribution of input and output. This distribution effectively forms a part of the problem specification, thus restricting the practical applicability of an algorithm.
- convenient design and analysis of algorithms – the computation can be described at the top level as a sequence of transformations of the global state determined by the contents of the shared memory. As a contrast, algorithms for distributed memory models have to be designed in terms of individual processors operating on their local memories.
- straightforward programming – the shared memory is uniformly accessible via single address space and two basic primitives: reading and writing. As a contrast, programming for distributed memory models is more complicated, typically involving point-to-point communication between processors via the network.

The computational model most widely used in the theory of parallel computing is the *Parallel Random Access Machine* (*PRAM*) (see e.g. [5, 12, 13, 18]). The PRAM consists of a potentially infinite number of processors, each connected to a common memory unit with potentially infinite capacity. The computation is completely synchronous. Accessing a single value in the memory costs the same as performing an arithmetic or Boolean operation on a single value.

Several variants of PRAM have been introduced. Among them are *exclusive read, exclusive write PRAM* (*EREW PRAM*), which requires that every memory cell is accessed by not more than one processor in any one step, and *concurrent read, concurrent write PRAM* (*CRCW PRAM*), which allows several processors to access a cell concurrently in one step. For CRCW PRAM, a rule for resolving concurrent writing must be adopted. One of the possibilities, realised in *combining CRCW PRAM* (see e.g. [5, pp. 690–691]), is to write some specified combination of the values being written and (optionally) the value stored previously at the target cell. A typical choice of the combining function is some commutative and associative operator such as the sum or the maximum of the values.

Another major model of parallel computation is the circuit model (see e.g. [13, 18]). A *circuit* is a directed acyclic graph (*dag*) with terminal nodes labeled as *constant*,

input or *output*, and nonterminal nodes labeled by arithmetic or Boolean operations. Algorithms that can be represented as circuits are oblivious, i.e. perform the same sequence of operations for any input (although the arguments and results of individual operations may, of course, depend on the inputs). Such algorithms are simpler to analyse than non-oblivious ones. Circuits also provide a useful intermediate stage in the design of algorithms for PRAM-type models: the problem of designing a circuit is separated from the problem of scheduling its underlying dag. For example, while the question of an optimal solution to the matrix multiplication problem remains open, one can find optimal scheduling for particular circuits representing the standard $\Theta(n^3)$ method, or the Strassen's $\Theta(n^{\log 7})$ method. In this paper we study the scheduling problem for several classes of dags.

Both the PRAM and the circuit model are simple and straightforward. However, these models do not take into account the limited computational resources of existing computers, and therefore are far from being realistic. The first step in making them more realistic was to introduce a new complexity measure, *efficiency*, depending on the number of processors used by the algorithm (see [14]). New parallel models were gradually introduced to account for resources other than the number of processors. Currently, dozens of such models exists; see [16, 17, 23] for their survey. Among the computer resources measured by these models are, according to [16], the number of processors, memory organisation (distributed or shared), communication latency, degree of asynchrony, bandwidth, message handling overhead, block transfer, memory hierarchy, memory contention, network topology, and many others.

Models that include many different resource metrics tend to be too complex. A useful model should be concise and concentrate on a small number of crucial resources. One of the simplest and most elegant parallel models is the BSP model – see [27, 18, 19] for the description of BSP as an emerging paradigm for general-purpose parallel computing. The BSP model is defined by a few qualitative characteristics: uniform network topology, barrier-style bulk synchronisation, – and three quantitative parameters: the number of processors, communication throughput and latency. The main principle of BSP is to regard communication and synchronisation as separate activities, possibly performed by different mechanisms. The corresponding costs are independent and compositional, i.e. can be simply added together to obtain the total cost. It is easy to extend the BSP model to account for memory efficiency as well.

In this paper we propose a variant of BSP, called BSPRAM, intended to support shared-memory style BSP programming. The memory of BSPRAM has two levels: local memory of individual processors, and a shared global memory. We compare BSPRAM with similar existing models. We then study the relationship between BSPRAM with BSP by means of simulation. Let n denote the size of the input to a program. Following [28], we say that a model A can *optimally simulate* a model B when there is a compilation algorithm that transforms any program with cost $T(n)$ on B to a program with cost $O(T(n))$ on A. If the compilation algorithm yields a randomised program for A, we call the simulation optimal if the expected cost of the randomised program is $O(T(n))$. Sometimes the simulation may be restricted to programs from a particular

class. We assume that we are free to define a suitable distribution of the input and output data to simulate a shared memory model on a distributed memory one.

If the described compilation is defined only for a particular class of algorithms, we say that A can optimally simulate B for that class of algorithms. We show that BSP can optimally simulate BSPRAM for several large classes of algorithms.

3. The BSP model

A *BSP computer*, introduced in [26–28], consists of p *processors* connected by a communication network (see Fig. 1). Each processor has a fast *local memory*. The processors may follow different threads of computation. A BSP computation is a sequence of *supersteps* (see Fig. 2). A superstep consists of an *input phase*, a *local 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 output phase it can send data to other processors, to be received in the next superstep. The processors are synchronised between supersteps. The computation within a superstep is 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.

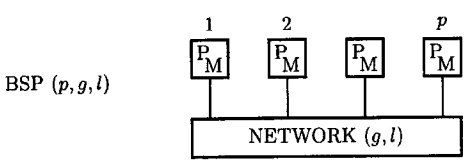

Fig. 1. A BSP computer.

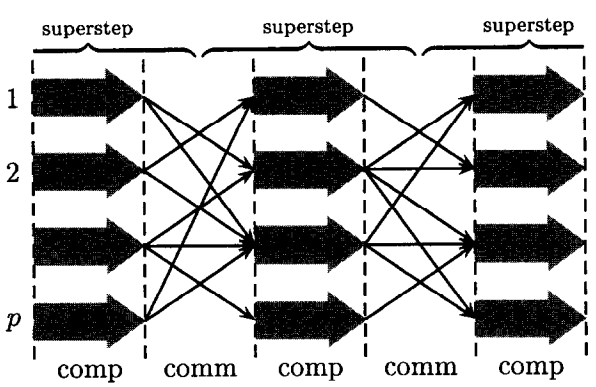

Fig. 2. A BSP computation.

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) to denote a BSP computer 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 $w_s + h_s \cdot g + l$, $1 \leqslant s \leqslant S$, then its total cost is $W + H \cdot g + S \cdot l$, where $W = \sum_{s=1}^{S} w_s$ is the local computation cost, $H = \sum_{s=1}^{S} h_s$ is the communication cost, and S is the synchronisation cost. (We will omit the factors g and l when dealing with communication and synchronisation separately from local computation.)

In order to utilise the computer resources efficiently, a typical BSP program should regard the values p, g and l as configuration parameters. Algorithm design should aim to minimise local computation, communication and synchronisation costs for any realistic values of these parameters. For most problems, a balanced distribution of data and computation work will lead to algorithms that achieve optimal cost values simultaneously. However, for some other problems a need to trade off the costs will arise.

An example of a communication-synchronisation tradeoff is the problem of broadcasting a single value from a processor: it can be performed with $H = S = \mathrm{O}(\log p)$ by a balanced binary tree, or with $H = \mathrm{O}(p)$ and $S = \mathrm{O}(1)$ by sending the value directly to every processor (this was observed in [27]). On the other hand, a technique known as *two-phase broadcast* allows one to achieve perfect balance for the problem of broadcasting $n \geqslant p$ values from one processor. By dividing the values into p blocks of size n/p, scattering the blocks so that each one gets to a distinct processor, and then performing total exchange of the blocks, the problem can be solved with $H = \mathrm{O}(n)$ and $S = \mathrm{O}(1)$ – this is obviously optimal. Broadcasting of $n = p^{\varepsilon}$ elements for any constant ε, $0 < \varepsilon < 1$, can be performed optimally by $(1 + \varepsilon^{-1})$-*phase broadcast*. The values are scattered so that each one gets to a distinct processor, and each value is broadcast by a balanced tree of degree n and height ε^{-1}. Non-leaf nodes of the broadcasting forest are partitioned among the processors, so that on each level each processor computes at most one node. The communication and synchronisation costs are $H = \mathrm{O}(\varepsilon^{-1} \cdot n) = \mathrm{O}(n)$ and $S = \mathrm{O}(\varepsilon^{-1}) = \mathrm{O}(1)$.

Matrix computations provide further examples of problems with and without tradeoffs: for instance, matrix multiplication can be done optimally in communication and synchronisation, but matrix inversion presents a tradeoff between communication and synchronisation with a polynomial range of parameters.

The BSP model does not directly support shared memory, broadcasting or combining. These facilities can be obtained by simulating a PRAM on a BSP computer. Such simulation is also called *automatic mode* BSP programming, as opposed to the *direct mode*, i.e. programming with explicit control over communication and synchronisation.

In order to achieve efficient simulation of a PRAM on a BSP computer, the PRAM must have more processors than the BSP computer. For a fixed value of p, we say that a PRAM algorithm has *slackness* σ, if at least σp PRAM processors perform reading or writing at every step. Note that σp is a lower bound on the number of

communicating processors, rather than the actual minimum number, which may depend on the dynamic behaviour of the algorithm. Slackness measures the "degree of communication parallelism" achieved by the algorithm, and is typically a function of the problem size n and the number of BSP processors p.

In the automatic mode, each step of a PRAM is implemented as a superstep, with at least σ virtual PRAM processors allocated to each of the p BSP processors. Virtual processor allocation is equal and non-repeating, but otherwise arbitrary. Paper [28] states the following result.

Theorem 1. *An optimal randomised simulation on BSP* (p,g,l) *can be achieved for*
(i) *any EREW PRAM algorithm with slackness* $\sigma \geqslant \log p$;
(ii) *any CRCW PRAM algorithm with slackness* $\sigma \geqslant p^{\varepsilon}$ *for some* $\varepsilon > 0$.
Here g and l are assumed to be constant.

Proof. See [28]. □

Memory access in the randomised simulation is made uniform by *hashing*: each memory cell of the simulated PRAM is represented by a cell in the local memory of one of the BSP processors, chosen according to some easily computable *hash function* which ensures nearly random and independent distribution of cells.

The simulation allows one to write PRAM programs for BSP computers and to predict their performance accurately. Most practical problems possess the slackness necessary for efficient simulation. However, the automatic mode does not allow the programmer to exploit data locality, because PRAM processors do not have any substantial local memory. For some problems this is insignificant (e.g. multiplication of sparse matrices with a random pattern of nonzeros). For many other problems this can be a serious drawback. (e.g. multiplication of dense or regularly sparse matrices). Because of that, the direct mode of BSP programming is often preferable to the automatic mode.

The next section aims to reconcile the exploitation of data locality with shared-memory style programming, retaining the parameters g and l and the bulk-synchronous structure of the computation. We introduce a new BSP-type model, called BSPRAM, in which the network is implemented as a random-access shared memory unit. The new model is designed to combine the best features of both automatic and direct BSP programming modes. We present a randomised BSP simulation of BSPRAM, based on a suitably adapted concept of slackness. We also describe a deterministic simulation, based on additional properties of obliviousness and granularity.

4. The BSPRAM model

In the previous section we described two alternative approaches to BSP programming. The automatic mode (PRAM simulation) enables the shared-memory style BSP programming with all its benefits. However, it does not allow one to exploit data

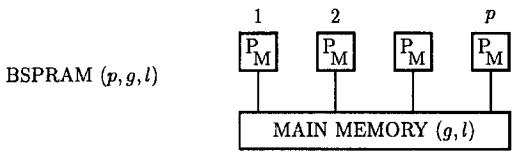

Fig. 3. A BSPRAM.

locality. On the other hand, the direct mode (pure BSP) allows one to exploit data locality, but only in a distributed memory paradigm. The aim of this section is to introduce a new BSP programming method, allowing both shared-memory style programming and exploitation of data locality. This might be called a "semi-automatic mode" of BSP programming.

The new method is similar to PRAM simulation mentioned in the previous section. The key difference is that a BSP superstep is no longer fragmented into independent steps of σp individual virtual PRAM processors. The structure of computation in the local memories of BSP processors is preserved. The simulation mechanism is used to model the global shared memory, which in the new model replaces the BSP communication network. We call the new computational model *BSPRAM*.

Formally, a BSPRAM consists of p *processors* with fast *local memories* (see Fig. 3). In addition, there is a single shared *main memory*. As in BSP, the computation proceeds by *supersteps* (see Fig. 2). A superstep consists of an *input phase*, a *local computation phase*, and an *output phase*. In the input phase a processor can read data from the main memory; in the output phase it can write data to the main memory. The processors are synchronised between supersteps. The computation within a superstep is asynchronous.

Similarly to PRAM, concurrent access to the main memory in one superstep can be either allowed or disallowed. In this paper we consider an *exclusive-read, exclusive-write BSPRAM* (*EREW BSPRAM*), in which every cell 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*), that 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 cell containing the value y, the result may be determined by any prescribed function $f(x, y)$ computable in time O (1). Similarly, if values $x_1, \ldots, x_m$ are being written concurrently to a main memory cell containing the value y, the result may be determined by any prescribed function $f(x_1 \oplus \cdots \oplus x_m, y)$, where $\oplus$ is a commutative and associative operator, and both f and $\oplus$ are computable in time O(1). This corresponds to resolving concurrent writing in PRAM by combining (see e.g. [5]).

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) to denote a BSPRAM with the given values

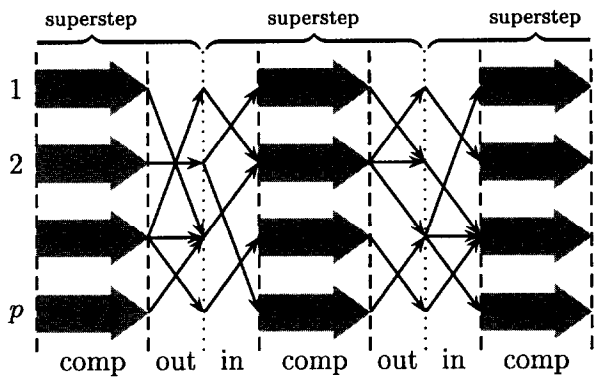

Fig. 4. A BSPRAM computation.

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.

One of the early models similar to BSPRAM was the LPRAM model proposed in [1]. The model consists of a number of synchronously working processors with large local memories and a global shared memory. The only mode of concurrent memory access considered in [1] is CREW. The model has an explicit bandwidth parameter, which corresponds to g in BSP and BSPRAM. There is no accounting for synchronisation cost (although it is suggested as a possible extension of the model). Thus, a p-processor LPRAM is equivalent (up to a constant factor) to CREW BSPRAM $(p, g, 1)$.

Another model similar to BSPRAM, called Asynchronous PRAM, was proposed in [7] (an earlier version of this model was called Phase PRAM). Like BSPRAM, Asynchronous PRAM consists of processor-memory pairs communicating via a global shared memory. The computation structure is bulk-synchronous, with EREW communication. The model charges a unit cost for a global read/write operation, d units for communication startup and B units for barrier synchronisation. Thus, a p-processor Asynchronous PRAM is equivalent (up to a constant factor) to EREW BSPRAM $(p, 1, d+B)$.

A bulk-synchronous parallel model QSM is proposed in [8] (an earlier version of this model was called QRQW PRAM). The model has a bandwidth parameter g. A p-processor QSM machine is similar to BSPRAM $(p, g, 1)$ with a special mode of concurrent access to the main memory: any k concurrent accesses to a cell cost k units. Such a model is more powerful than EREW BSPRAM $(p, g, 1)$, but less powerful than CRCW BSPRAM $(p, g, 1)$.

As for PRAM simulation, some "extra parallelism" is necessary for efficient BSPRAM simulation on BSP. We say that a BSPRAM algorithm has *slackness* σ, if the communication cost of every one of its supersteps is at least σ. We adapt the results on PRAM simulation mentioned in the previous section to obtain an efficient simulation of BSPRAM.

Theorem 2. *An optimal randomised simulation on BSP* (p,g,l) *can be achieved for*
(i) *any EREW BSPRAM* (p,g,l) *algorithm with slackness* $\sigma \geqslant \log p$;
(ii) *any CRCW BSPRAM* (p,g,l) *algorithm with slackness* $\sigma \geqslant p^{\varepsilon}$ *for some* $\varepsilon > 0$.

Proof. Immediately follows from Theorem 1. □

Apart from randomised simulation by hashing, in some cases an efficient deterministic simulation of BSPRAM is possible. We consider two important classes of algorithms for which such deterministic simulation exists.

We say that a BSPRAM algorithm is *oblivious*, if the sequence of operations executed by each processor is the same for any input of a given size (although the arguments and results of individual operations may depend on the inputs). An oblivious algorithm can be represented as a computation of a uniform family of circuits (for the definition of a uniform family of circuits, see e.g. [13]). We say that a BSPRAM algorithm is *communication-oblivious*, if the sequence of communication and synchronisation operations executed by each processor is the same for any input of a given size, but no such restriction is made for local computation.

We say that a set of cells in the main memory of BSPRAM constitutes a *granule*, if in any input (output) phase each processor either does not read from (write to) any of these cells, or reads from (writes to) all of them. Informally, a granule is treated as "one whole piece of data". We say that a BSPRAM algorithm has *granularity* γ if all main memory cells used by the algorithm can be partitioned into granules of size at least γ. Note that both slackness and granularity are lower bounds rather than the actual minimum values. Slackness of a BSPRAM algorithm can always be taken to be equal or higher than its granularity: $\sigma \geqslant \gamma$.

Communication-oblivious algorithms and algorithms with sufficient granularity for BSPRAM allow optimal deterministic BSP simulation. As we show below, randomised hashing is not necessary for communication-oblivious algorithms, since their communication pattern is known in advance. Therefore, an optimal distribution of main memory cells across BSP processor-memory pairs can be found off-line. For algorithms with granularity at least p, hashing is not necessary either, since every granule can be split up into p equal parts that are evenly distributed across BSP processor-memory pairs. This makes all communication uniform. In both cases randomised hashing is replaced by a simple deterministic data distribution. Moreover, for communication-oblivious algorithms with slackness at least p^{ε}, and for algorithms with granularity at least p, concurrent memory access can be simulated by mechanisms similar to the two-phase and $(1+\varepsilon^{-1})$-phase broadcast described in the previous section.

Below we formally state the results on deterministic BSPRAM simulation.

Theorem 3. *An optimal deterministic simulation on BSP* (p,g,l) *can be achieved for*
(i) *any communication-oblivious EREW BSPRAM* (p,g,l) *algorithm*;

(ii) *any communication-oblivious CRCW BSPRAM* (p,g,l) *algorithm with slackness* $\sigma \geqslant p^{\varepsilon}$ *for some* $\varepsilon > 0$;
(iii) *any CRCW BSPRAM* (p,g,l) *algorithm with granularity* $\gamma \geqslant p$.

Proof. (i) Since the communication pattern of a communication-oblivious algorithm is known in advance, we only need to show that any computation of EREW BSPRAM (i.e. a particular run of an algorithm) can be performed in BSP at the same asymptotic cost. First, we modify each BSPRAM superstep so that each processor both reads and writes any main memory cell that it either reads or writes in the original superstep. This increases the communication cost of the computation at most by a factor of 2, and does not change the synchronisation cost.

The above modification essentially transforms the computation into a form of message passing, in which main memory cells represent messages, and writing or reading a value corresponds to sending or receiving a message. This message-passing version of BSPRAM was referred to as "BSP+" in [25]. It differs from the direct BSP mode in that a message can be "delayed", i.e. its sending and receiving may occur in non-adjacent supersteps.

It remains to show that the "delayed" messages can be simulated optimally by normal BSP messages. We represent the whole BSPRAM computation by an undirected graph. Each superstep is represented by two nodes, one for the input phase and the other for the output phase. Messages are represented by edges. Two nodes v_1 and v_2 are connected by an edge e, if the message represented by e is sent in the output phase represented by v_1, and received in the input phase represented by v_2. The constructed graph is bipartite, with the two parts representing all input and output phases respectively. If an input or output phase has cost h, then the degree of its representing node is at most ph.

It is a well-known fact (see e.g. [2, p. 247]), that for any bipartite graph with maximum degree at most p, there is a colouring of its edges with not more than p colours, such that all the edges adjacent to the same node are coloured differently. As an easy corollary of this, for an arbitrary bipartite graph and an arbitrary p, there is a colouring of the edges with not more than p colours, such for an arbitrary h, any node of degree at most ph has at most h adjacent edges of of each colour. (This can be proved by splitting each node of degree at most ph into h nodes of degree at most p.)

We use the above theorem to colour the computation graph. We then regard the colour of each edge as the identifier of a BSP processor that must obtain the corresponding message from the sending processor, keep it in its local memory for as long as necessary, and then transfer the message to the receiving processor. The communication and synchronisation costs of the computation are increased at most by a factor of 2.

(ii) The proof is similar to that of (i). The only difference is that, due to concurrent reading and writing, each message has to be combined from contributions of several processors before being sent, and broadcast to several processors after being

received. Consider a particular superstep in the computation. By symmetry, we need to analyse only the input phase. Simultaneous broadcasting of received messages is done by a method which generalises the $(1+\varepsilon^{-1})$-phase broadcast technique from Section 3. Without loss of generality we assume that the communication cost of the considered input phase is $h=\sigma=p^{\varepsilon}$, $0<\varepsilon<1$. Each message is broadcast by a tree of maximum degree h and height at most ε^{-1} (the tree does not have to be balanced). The broadcasting forest is partitioned among the processors so that on each level the total degree of nodes computed in any processor is at most $2h$. Such partitioning can be easily obtained by a greedy algorithm. The communication cost of the computation is increased at most by a factor of $2\varepsilon^{-1}$, and the synchronisation cost at most by a factor of ε^{-1}.

(iii) Partition each granule into p equal subgranules. For each granule, choose an arbitrary balanced distribution of its subgranules across the processors.

An input phase of the BSPRAM algorithm is simulated by two BSP supersteps. In the first superstep a processor broadcasts a request for each granule that it must read. Note that since the subgranules of every granule are distributed evenly, all processors receive an identical set of requests. In the second superstep a processor satisfies the received requests by sending the locally stored subgranules of the requested granules to the requesting processors.

An output phase of the BSPRAM algorithm is simulated by one BSP superstep. In this superstep a processor divides each granule that it must write into p subgranules, and sends to every processor the appropriate subgranules. Having received its subgranules, each processor combines any concurrently written data, and then updates the locally stored subgranules.

The communication and synchronisation costs of the computation are increased at most by a factor of 2. □

On some parallel computers, a direct implementation of the BSPRAM model may prove practical. In any case, the proofs of Theorems 2 and 3 show that a BSP computer can execute most practical BSPRAM algorithms within a low constant factor of their BSPRAM cost. For two important classes of algorithms – communication-oblivious algorithms and algorithms with sufficient granularity – the simulation is deterministic and particularly simple. The next few sections give examples of such algorithms.

5. Butterfly dag computation in BSPRAM

The butterfly dag describes the dependence pattern of the Fast Fourier Transform, which is one of the most important algorithms in scientific computation. Parallel algorithms for butterfly dag computation have been proposed in various parallel models (see e.g. [5, 12]).

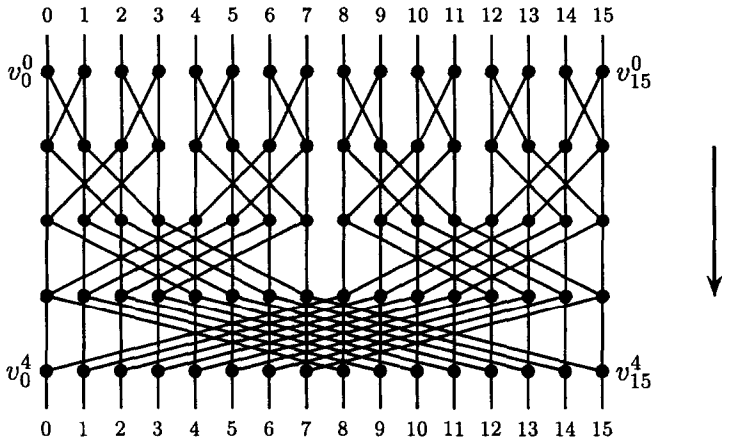

Fig. 5. Butterfly dag *bfly*(16).

Formally, the *butterfly dag* *bfly*(n) with inputs x_i and outputs y_i, $0 \leqslant i < n$, contains $\log n+1$ levels of nodes v_i^k, $0 \leqslant k \leqslant \log n$, $0 \leqslant i < n$, such that

$$\begin{aligned} &v_i^0 \text{ takes the input } x_i, \\ &v_j^{k+1} \text{ depends on } v_i^k, \text{if } i=j \text{ or } i \oplus j = 2^k, \\ &v_i^{\log n} \text{ produces the output } y_i, \end{aligned} \tag{1}$$

where $i \oplus j$ denotes the bitwise x or (exclusive or) operation on the binary representations of i and j. Fig. 5 shows the butterfly dag *bfly*(16).

As observed in [21, 27], the butterfly dag can be partitioned in a way suitable for bulk-synchronous parallel computation. The computation of a level in *bfly*(n) consists in $\frac{1}{2}n$ independent computations of *bfly*(2). Similarly, the computation of any k consecutive levels consists in $n/2^k$ independent computations of *bfly*(2^k). Therefore, the butterfly dag computation can be split into two stages, each comprising $\frac{1}{2}\log n$ levels and consisting of $n^{1/2}$ independent tasks. If n is sufficiently large with respect to p, each of the two stages can be completed in one superstep.

Fig. 6 shows the two-superstep computation of *bfly*(16). Each superstep consists of four independent tasks computing *bfly*(4). In general, the algorithm is as follows.

Algorithm 1. *Computation of the butterfly dag* *bfly*(n).

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

Output: An array $\mathbf{y}=(y_i)$, $0 \leqslant i < n$, defined by (1).

Description. We assume $n \geqslant p^2$. The computation is performed on EREW BSPRAM (p, g, l) and proceeds in two supersteps, each comprising $\frac{1}{2}\log n$ levels. In both supersteps, each processor is assigned $n^{1/2}/p$ independent butterfly dags of size $n^{1/2}$. Data are communicated via the main memory.

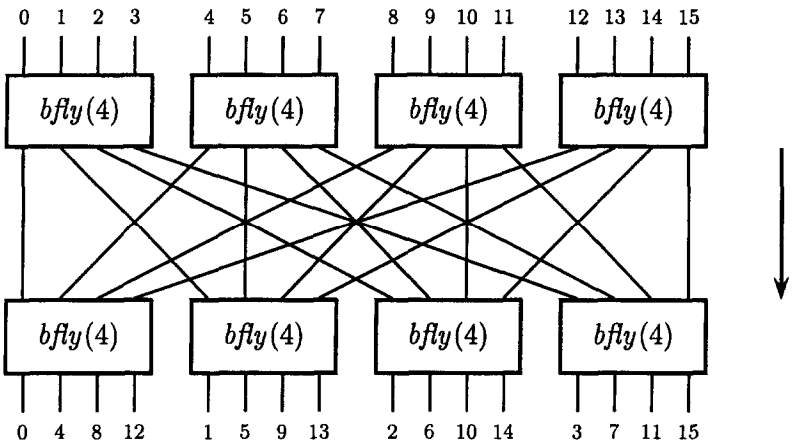

Fig. 6. BSPRAM computation of *bfly*(16).

Cost analysis. The local computation, communication and synchronisation costs are

$$W = O(n \log n/p), \qquad H = O(n/p), \qquad S = O(1).$$

The algorithm is oblivious, with slackness $\sigma = n/p$, and granularity $\gamma = n/p^2$.

Paper [27] considers reducing the required minimum value of n for efficient BSP computation of a butterfly dag.

6. Cube dag computation in BSPRAM

The cube dag defines the dependence pattern that is characteristic for many scientific algorithms. Here we describe a BSPRAM version of the BSP cube dag algorithm from [19]. For simplicity, we consider the computation of a three-dimensional cube dag; the algorithm for other dimensions is similar.

The three-dimensional cube dag $cube_3(n)$ with inputs $x^{(1)}_{jk}$, $x^{(2)}_{ik}$, $x^{(3)}_{ij}$, and outputs $y^{(1)}_{jk}$, $y^{(2)}_{ik}$, $y^{(3)}_{ij}$, $0 \leqslant i,j,k < n$, contains n^3 nodes v_{ijk}, such that

$$\begin{aligned} &v_{0jk}, v_{i0k}, v_{ik0} \text{ take respectively } x^{(1)}_{jk}, x^{(2)}_{ik}, x^{(3)}_{ij} \\ &v_{ijk} \text{ contributes to each of the nodes} \\ &\qquad v_{i+1,j,k}, v_{i,j+1,k}, v_{i,j,k+1} \text{ whenever such node exists} \\ &v_{n-1,j,k}, v_{i,n-1,k}, v_{i,j,n-1} \text{ produce respectively } y^{(1)}_{jk}, y^{(2)}_{ik}, y^{(3)}_{ij} \end{aligned} \tag{2}$$

Fig. 7 shows the cube dag $cube_3(4)$.

The BSP algorithm for computing the dag $cube_3(n)$ is given in [19]. In this algorithm, the array $V = (v_{ijk})$ is partitioned into $p^{3/2}$ regular cubic blocks of volume $(n/p^{1/2})^3$. We denote these blocks by V_{ijk}, $0 \leqslant i,j,k < p^{1/2}$. Each block defines a dag isomorphic to $cube_3(n/p^{1/2})$. The algorithm computes a block $V_{i+1,j+1,k+1}$ as soon as the data from

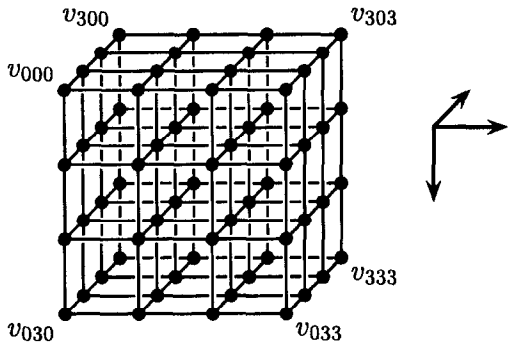

Fig. 7. Cube dag $cube_3(4)$.

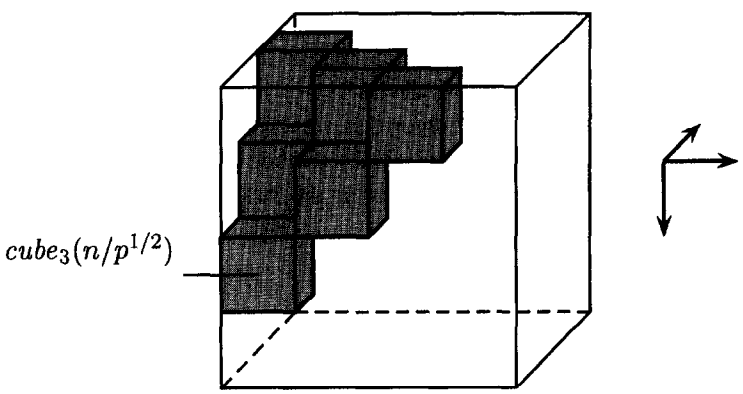

Fig. 8. BSPRAM computation of $cube_3(n)$.

its predecessors $V_{i,j+1,k+1}$, $V_{i+1,j,k+1}$ and $V_{i+1,j+1,k}$ become available. The independent blocks computed simultaneously form a "layer", or "wavefront" of the dag $cube_3(n)$.

Fig. 8 shows a stage in the BSP computation of $cube_3(n)$. The current wavefront is shaded. The total number of wavefronts is $3p^{1/2}-2$, therefore the computation can be completed in $O(p^{1/2})$ supersteps.

Algorithm 2. *Computation of the cube dag* $cube_3(n)$

Input: Arrays $\boldsymbol{x}^{(1)}=(x^{(1)}_{jk})$, $\boldsymbol{x}^{(2)}=(x^{(2)}_{ik})$, $\boldsymbol{x}^{(3)}=(x^{(3)}_{ij})$, $0\leqslant i,j,k<n$.

Output: Arrays $\boldsymbol{y}^{(1)}=(y^{(1)}_{jk})$, $\boldsymbol{y}^{(2)}=(y^{(2)}_{ik})$, $\boldsymbol{y}^{(3)}=(y^{(3)}_{ij})$, $0\leqslant i,j,k<n$, defined by (2).

Description. We assume $n\geqslant p^{1/2}$. The computation is performed on EREW BSPRAM (p,g,l) and proceeds in $3p^{1/2}-2$ stages, each comprising a constant number of supersteps. In stage s, $0\leqslant s<3p^{1/2}-3$, the blocks V_{ijk} with $i+j+k=s$ are computed. The maximum number of blocks computed in any one stage is $\frac{3}{4}p$. Data are communicated between supersteps via the main memory.

Cost analysis. The local computation cost is $W=O(n^3/p)$. The computation of a block requires the communication (reading from or writing to the main memory) of

$O(n^2/p)$ values on the surface of the block. Therefore, the communication cost of each stage is $h = O(n^2/p)$. The total communication cost is $H = h \cdot p^{1/2} = O(n^2/p^{1/2})$. The synchronisation cost is $S = O(p^{1/2})$. The algorithm is oblivious, with slackness and granularity $\sigma = \gamma = n^2/p$.

7. Matrix multiplication in BSPRAM

In this section we describe a BSPRAM algorithm for one of the most common problems in scientific computation: dense matrix multiplication. We deal with the problem of computing the matrix product $XY = Z$, where $X = (x_{ij})$, $Y = (y_{jk})$, $Z = (z_{ik})$ are arbitrary $n \times n$ matrices.

This problem is of great importance, and, despite its simple formulation, of enormous theoretical complexity. Since the groundbreaking paper by Strassen [24] much work has been done on the complexity of sequential matrix multiplication. However, no lower bound asymptotically better than the trivial $\Omega(n^2)$ has been found; nor there is any indication that the current $O(n^{2.376})$ algorithm from [4] is close to optimal.

We aim at parallelising the standard $\Theta(n^3)$ method without using fast matrix multiplication techniques. The method consists in the straightforward computation of the family of bilinear forms

$$z_{ik} = \sum_{j=1}^{n} x_{ij} y_{jk}, \quad 1 \leqslant i, k \leqslant n. \tag{3}$$

Following (3), we need to set

$$z_{ik} \leftarrow 0 \quad \text{for } i, k = 1, \ldots, n \tag{4}$$

and then compute

$$v_{ijk} \leftarrow x_{ij} y_{jk}, \qquad z_{ik} \leftarrow z_{ik} + v_{ijk} \quad \text{for all } i, j, k, \ 1 \leqslant i, j, k \leqslant n. \tag{5}$$

Computation (5) for different triples i, j, k is independent (although it requires concurrent reading from x_{ij} and y_{jk}, and concurrent writing to z_{ik}), and therefore can be performed in parallel.

The BSPRAM algorithm implementing this method is derived from the BSP algorithm for matrix multiplication described in [19, 20], which in its turn is based on an idea from [1]. The algorithm works by a straightforward partitioning of the problem. The array $V = (v_{ijk})$ is represented as a cube of volume n^3 in integer three-dimensional space (see Fig. 9). The arrays X, Y, Z are represented as projections of the cube V onto the coordinate planes $k = 0$, $i = 0$ and $j = 0$, respectively. The computation with the point v_{ijk} in (5) requires the input of its X and Y projections x_{ij} and y_{jk}, and the output of its Z projection z_{ik}. In order to provide a communication-efficient BSP algorithm, the array V must be divided into p regular cubic blocks of size $n/p^{1/3}$ (see Fig. 10). Such partitioning induces a partition of the matrices X, Y and Z into $p^{2/3}$

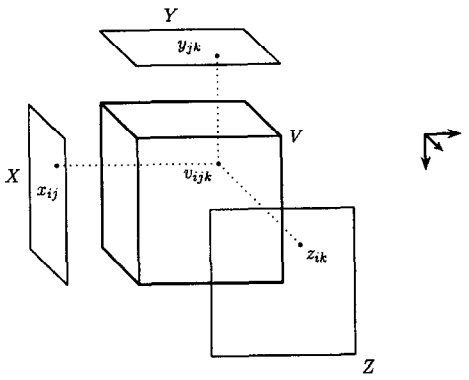

Fig. 9. Matrix multiplication dag.

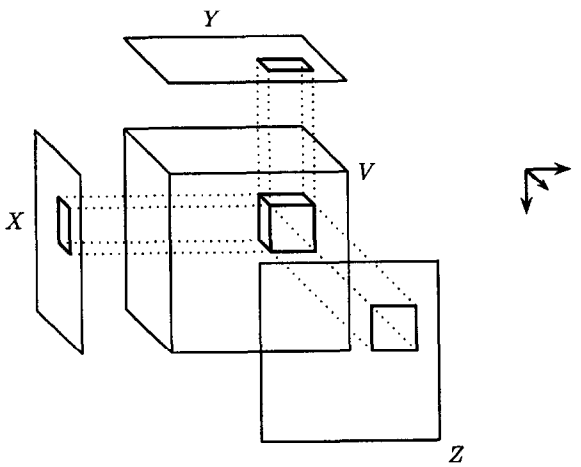

Fig. 10. Matrix multiplication in BSPRAM.

regular square blocks of size $n/p^{1/3}$,

$$X = \begin{pmatrix} X_{11} & \cdots & X_{1,p^{1/3}} \\ \vdots & \ddots & \vdots \\ X_{p^{1/3},1} & \cdots & X_{p^{1/3},p^{1/3}} \end{pmatrix} \tag{6}$$

and similarly for Y and Z (see Fig. 10). The computation (4), (5) can be expressed in terms of blocks as

$$Z_{ik} \leftarrow 0 \quad \text{for } i,k = 1,\ldots,p^{1/3} \tag{7}$$

and then

$$V_{ijk} \leftarrow X_{ij} Y_{jk}, \quad Z_{ik} \leftarrow Z_{ik} + V_{ijk}, \quad \text{for all } i,j,k, \ 1 \leqslant i,j,k \leqslant p^{1/3}. \tag{8}$$

Each processor computes a block product $V_{ijk} = X_{ij} \cdot Y_{jk}$ sequentially by (4), (5). The algorithm is as follows.

Algorithm 3. *Matrix multiplication.*

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

Output: A matrix $Z = (z_{ij})$, $1 \leqslant i, j \leqslant n$, defined by (3).

Description. We assume $n \geqslant p^{1/3}$. The computation is performed on CRCW BSPRAM (p, g, l).

After the initialisation step (7), the computation proceeds in one superstep. Each processor performs the computation (8) for a particular triple i, j, k. In the input phase, the processor reads X_{ij} and Y_{jk}. Then it computes the product $V_{ijk} = X_{ij} \cdot Y_{jk}$ by (4), (5). The block V_{ijk} is then written to Z_{ik} in the main memory. Concurrent writing is resolved by addition of the written blocks to the previous content of Z_{ik}. The resulting array Z is the matrix product of X and Y.

Cost analysis. The local computation, communication and synchronisation costs are

$$W = \mathrm{O}(n^3/p), \qquad H = \mathrm{O}(n^2/p^{2/3}), \qquad S = \mathrm{O}(1).$$

The algorithm is oblivious, with slackness and granularity $\sigma = \gamma = n^2/p^{2/3}$.

8. Sorting in BSPRAM

Sorting is a classical problem of parallel computing. Many parallel sorting algorithms of different complexity have been proposed (see e.g. [3, 9, 12] and references therein). Here we consider comparison-based sorting of an array $\boldsymbol{x} = (x_i)$, $1 \leqslant i \leqslant n$. Without loss of generality we may assume that the elements of $\boldsymbol{x}$ are distinct (otherwise, we should attach a unique tag to each element). Let $\langle a, b \rangle$ denote an *open interval*, i.e. the set of all x in $\boldsymbol{x}$ such that $a < x < b$.

Probably the simplest parallel sorting algorithm is parallel sorting by regular sampling (PSRS), proposed in [22] and discussed in [15]. Paper [11] describes an optimised version of the algorithm, and its efficient implementation on a variety of platforms.

The PSRS algorithm proceeds as follows. First, the array $\boldsymbol{x}$ is partitioned into p subarrays $\boldsymbol{x}^1, \ldots, \boldsymbol{x}^p$, each of size n/p. The subarrays $\boldsymbol{x}^q$ are sorted independently by an optimal sequential algorithm. The problem now consists in merging the p sorted subarrays.

On the first stage of merging, $p + 1$ regularly spaced *primary samples* are selected from each subarray (the first and the last elements of a subarray are among the samples). We denote the samples of the subarray $\boldsymbol{x}^q$ by $\overline{x}_0^q, \ldots, \overline{x}_p^q$. The samples divide each subarray into p *primary blocks* of size n/p^2. We denote the primary blocks of $\boldsymbol{x}^q$ by $[\overline{x}_0^q, \overline{x}_1^q], \ldots, [\overline{x}_{p-1}^q, \overline{x}_p^q]$. Then, $p \cdot (p + 1)$ primary samples are collected together

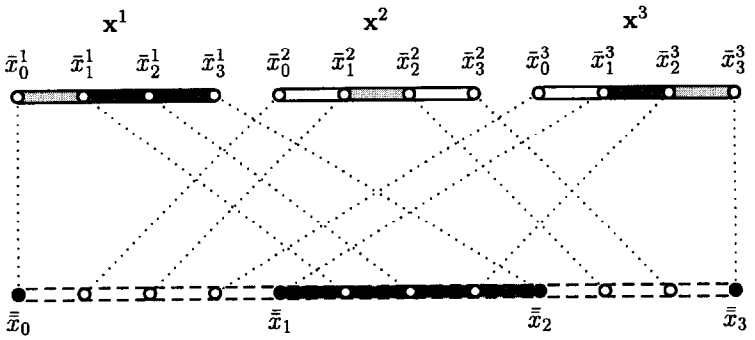

Fig. 11. Sorting by regular sampling on three processors.

and sorted by an arbitrary sequential algorithm. After that, we choose $p+1$ regularly spaced *secondary samples* from the sorted array of primary samples (the first and the last elements are again included in the samples). We denote the secondary samples by $\bar{\bar{x}}_0,\ldots,\bar{\bar{x}}_p$. The secondary samples partition the elements of $\boldsymbol{x}$ into p *secondary blocks*, corresponding to the intervals $\langle\bar{\bar{x}}_0,\bar{\bar{x}}_1\rangle,\ldots,\langle\bar{\bar{x}}_{p-1},\bar{\bar{x}}_p\rangle$. Efficiency of the above computations with samples is not critical, since the number of samples does not depend on n. The problem is now reduced to collecting the elements of each secondary block together.

Let us show that each secondary block contains at most $3n/p$ elements. For a fixed secondary block defined by $\langle\bar{\bar{x}}_k,\bar{\bar{x}}_{k+1}\rangle$, we divide all the primary blocks of $\mathbf{x}$ into three categories. We call a primary block $\langle\bar{x}_i^q,\bar{x}_{i+1}^q\rangle$ an *inner block*, if $\langle\bar{x}_i^q,\bar{x}_{i+1}^q\rangle\subseteq\langle\bar{\bar{x}}_k,\bar{\bar{x}}_{k+1}\rangle$; an *outer block*, if $\langle\bar{x}_i^q,\bar{x}_{i+1}^q\rangle\cap\langle\bar{\bar{x}}_k,\bar{\bar{x}}_{k+1}\rangle=\emptyset$; and a *boundary block*, if it is neither inner nor outer. With respect to any secondary block, there are at most p inner primary blocks in total (because there are only p primary samples inside the secondary block) and at most two boundary primary blocks in each subarray (because a boundary block must contain at least one of the two ends of the secondary block). Therefore, the size of a secondary block is at most $n/p^2\cdot(p+2p)=3n/p$. Thus, on the second stage of merging, the elements of each secondary block can be collected in optimal time, and then sorted by an optimal sequential algorithm.

The method is illustrated in Fig. 11 for $p=3$. The state of the array $\boldsymbol{x}$ after local sorting of the subarrays is represented by three horizontal bars at the top. Primary samples are shown as white dots. Dotted lines show the rearrangement of primary samples into a sorted array at the bottom. The dashed bars at the bottom show the elements of $\boldsymbol{x}$ assumed to lie between the samples; their numbers between neighbouring primary samples need not be equal. Black dots indicate the secondary samples. The secondary block $\langle\bar{\bar{x}}_1,\bar{\bar{x}}_2\rangle$ is shown by dark shading. Primary blocks that are inner, boundary and outer for $\langle\bar{\bar{x}}_1,\bar{\bar{x}}_2\rangle$ are shown by dark shading, light shading and no shading, respectively. Only inner and boundary blocks may contain elements from $\langle\bar{\bar{x}}_1,\bar{\bar{x}}_2\rangle$.

The sorting algorithm based on PSRS can be easily implemented in the BSPRAM model. We assume that the input and output arrays are stored in the main memory of BSPRAM.

Algorithm 4. *Sorting by regular sampling.*

Input: an array $x=(x_i)$, $0 \leqslant i < n$, where all x_i are distinct.

Output: the elements of x in increasing order.

Description. We assume $n \geqslant p^3$. The computation is performed on CRCW BSPRAM (p, g, l) and proceeds in three supersteps. In the first superstep a processor picks a subarray x^q, reads it, sorts it with an optimal sequential algorithm, selects from it $p+1$ primary samples, and writes them to the main memory. In the second superstep the processors perform an identical computation: read the $p \cdot (p+1)$ primary samples, sort them and select p secondary samples. In the third superstep a processor picks a secondary block and collects its elements. In order to do this, a processor receives from other processors (via the main memory) all primary blocks that may intersect with the assigned secondary block; the number of such blocks is at most $3p$, and their total size is at most $3n/p$. The processor merges the primary blocks, discarding the values that do not belong to the assigned secondary block. The merged result is written to the main memory.

Cost analysis. The local computation, communication and synchronisation costs are

$$W = \mathrm{O}(n \log n/p) \qquad H = \mathrm{O}(n/p) \qquad S = \mathrm{O}(1)$$

The algorithm is not communication-oblivious. Its slackness and granularity are (ignoring non-critical computations with samples) $\sigma = n/p$, $\gamma = n/p^2$.

Paper [10] presents a more complex BSP sorting algorithm which is asymptotically optimal for any $n \geqslant p$. Its costs are $W = \mathrm{O}(n \log n/p)$, $H = \mathrm{O}(n/p \cdot \log n/\log(n/p))$, $S = \mathrm{O}(\log n/\log(n/p))$. For $n \geqslant p^3$, the algorithm is identical to PSRS; for smaller values of n it uses a pipelined tree merging technique similar to the one employed by Cole's algorithm (see e.g. [3]). Despite its asymptotic optimality, the algorithm from [10] is unlikely to be practical in the case of $n \approx p$. A more practical BSP sorting algorithm for small values of n is described in [6].

9. Conclusions

A new model for bulk-synchronous parallel computing, the BSPRAM, has been presented. The model enables the shared-memory style BSP programming with efficient exploitation of data locality. The BSP model can simulate BSPRAM optimally for a broad range of algorithms. The use of BSPRAM was illustrated on the examples of butterfly dag computation, cube dag computation, matrix multiplication and sorting. The corresponding values of the BSP cost, the type of BSPRAM used, and the characteristics of the obtained algorithms are summarised in Table 1. The BSPRAM approach

Table 1
Summary of algorithm examples (constant factors omitted)

Problem	W	H	S	type	c/obl?	σ	γ
bfly dag	$\frac{n \log n}{p}$	n/p	1	EREW	yes	n/p	n/p^2
cube dag	n^3/p	$\frac{n^2}{p^{1/2}}$	$p^{1/2}$	EREW	yes	n^2/p	n^2/p
matr mult	n^3/p	$\frac{n^2}{p^{2/3}}$	1	CRCW	yes	$\frac{n^2}{p^{2/3}}$	$\frac{n^2}{p^{2/3}}$
sorting	$\frac{n \log n}{p}$	n/p	1	CRCW	no	n/p	n/p^2

encourages natural specification of the problems: the input and output data are assumed to reside in the main memory, and no assumptions on data distribution are necessary. The design and analysis of algorithms are also simplified, since all communication is performed via the shared memory.

In future we plan to develop new BSPRAM algorithms and to analyse their costs. This may lead to identifying new algorithm properties connecting BSPRAM and BSP, in addition to obliviousness, slackness and granularity. We also plan to develop a programming model and an implementation of BSPRAM.

Acknowledgements

The author thanks Bill McColl, Karina Terekhova, Alex Gerbessiotis, Constantinos Siniolakis, Rob Bisseling, Ben Juurlink, and the anonymous referees.

References

[1] A. Aggarwal, A.K. Chandra, M. Snir, Communication complexity of PRAMs, Theoret. Comput. Sci. 71 (1990) 3–28.
[2] C. Berge, Graphs, vol. 6, part 1 of North-Holland Mathematical Library, 2nd rev. ed., North-Holland, Amsterdam, 1985.
[3] R. Cole, Parallel merge sort, in: J.H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann, Los Altos, CA, 1993, Ch. 10, pp. 453–495.
[4] D. Coppersmith, S. Winograd, Matrix multiplication via arithmetic progressions, J. Symbol. Comput. 9 (1990) 251–280.
[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, the MIT Electrical Engineering and Computer Science Series, MIT Press, Cambridge, MA and McGraw-Hill, New York, 1990.
[6] A.V. Gerbessiotis, C.J. Siniolakis, Deterministic sorting and randomized median finding on the BSP model, in: Proc. 8th ACM Symp. on Parallel Algorithms and Architectures, 1996, pp. 223–232.
[7] P.B. Gibbons, Asynchronous PRAM algorithms, in: J.H. Reif (Ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann, Los Altos, CA, 1993, Ch. 22, pp. 957–997.
[8] P. Gibbons, Y. Matias, V. Ramachandran, Can a shared memory model serve as a bridging model for parallel computation?, in: Proc. 9th ACM Symp. on Parallel Algorithms and Architectures, pp. 72–83, 1997, Extended abstract.
[9] A. Gibbons, W. Rytter, Efficient Parallel Algorithms, Cambridge University Press, Cambridge, 1988.
[10] M. Goodrich, Communication-efficient parallel sorting, in: Proc. 28th ACM Symp. on Theory of Computing, 1996.

[11] D.R. Helman, J. JáJá, D.A. Bader, A new deterministic parallel sorting algorithm with an experimental evaluation, Technical Report CS-TR-3670 and UMIACS-TR-96-54, Institute for Advanced Computer Studies, University of Maryland, August 1996.
[12] J. JáJá, An Introduction to Parallel Algorithms, Addison-Wesley, Reading, MA, 1992.
[13] R.M. Karp, V. Ramachandran, Parallel algorithms for shared memory machines, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Elsevier, Amsterdam, 1990, Ch. 17, pp. 869–941.
[14] C.P. Kruskal, L.Rudolph, M. Snir, A complexity theory of efficient parallel algorithms, Theoret. Comput. Sci. 71 (1990) 95–132.
[15] X. Li, P. Lu, J. Schaeffer, J. Shillington, P.S. Wong, Hanmao Shi, On the versatility of parallel sorting by regular sampling, Parallel Comput. 19 (1993) 1079–1110.
[16] Z. Li, P.H. Mills, J.H. Reif, Models and resource metrics for parallel and distributed computation, in: Proc. 28th Hawaii Internat. Conf. on System Sciences, IEEE Press, New York, 1995.
[17] B.M. Maggs, L.R. Matheson, R.E. Tarjan, Models of parallel computation: a survey and synthesis, in: Proc. 28th Hawaii Internat. Conf. on System Sciences, IEEE Press, New York, 1995, vol. 2, pp. 61–70.
[18] W.F. McColl, General purpose parallel computing, in: A. Gibbons, P. Spirakis (Eds.), Lectures on parallel computation, Cambridge International Series on Parallel Computation, vol. 4, Cambridge University Press, Cambridge, 1993, Ch. 13, pp. 337–391.
[19] W.F. McColl, Scalable computing, in: J. van Leeuwen (Ed.), Computer Science Today: Recent Trends and Developments, Lecture Notes in Computer Science, vol. 1000, Springer, Berlin, 1995, pp. 46–61.
[20] W.F. McColl, Universal computing, in: L. Bougé, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Proc. Euro-Par'96-I, Lecture Notes in Computer Science, vol. 1123, Springer, Berlin, 1996, pp. 25–36.
[21] C.H. Papadimitriou, M. Yannakakis, Towards an architecture-independent analysis of parallel algorithms, in: Proc. 20th Annual Symp. on Theory of Computing, 1988, pp. 510–513.
[22] Hanmao Shi, J. Schaeffer, Parallel sorting by regular sampling, Parallel Distrib. Comput. 14 (4) (1992) 361–372.
[23] D.B. Skillicorn, D. Talia, Models and languages for parallel computation, October 1996.
[24] V. Strassen, Gaussian elimination is not optimal, Numer. Math. 13 (1969) 354–356.
[25] A. Tiskin, The bulk-synchronous parallel random access machine, in: L. Bougé, P. Friagniaud, A. Mignotte, Y. Robert (Eds.), Proc. Euro-Par '96-II, Lecture Notes in Computer Science, vol. 1124, Springer, Berlin, 1996, pp. 327–338.
[26] L.G. Valiant, Bulk-synchronous parallel computers, in: M. Reeve (Ed.), Parallel Processing and Artificial Intelligence, Wiley, New York, 1989, Ch. 2, pp. 15–22.
[27] L.G. Valiant, A bridging model for parallel computation, Comm. ACM 33 (8) (1990) 103–111.
[28] L.G. Valiant, General purpose parallel architectures, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Elsevier, Amsterdam, Ch. 18, pp. 943–971.