The Design and Analysis of Bulk-Synchronous Parallel Algorithms

Alexandre Tiskin

Christ Church
Trinity Term 1998

Qualifying dissertation submitted
in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
at the University of Oxford

Programming Research Group
Oxford University Computing Laboratory

To the memory of my father

Abstract

The model of bulk-synchronous parallel (BSP) computation is an emerging paradigm of general-purpose parallel computing. This thesis presents a systematic approach to the design and analysis of BSP algorithms. We introduce an extension of the BSP model, called BSPRAM, which reconciles shared-memory style programming with efficient exploitation of data locality. The BSPRAM model can be optimally simulated by a BSP computer for a broad range of algorithms possessing certain characteristic properties: obliviousness, slackness, granularity. We use BSPRAM to design BSP algorithms for problems from three large, partially overlapping domains: combinatorial computation, dense matrix computation, graph computation. Some of the presented algorithms are adapted from known BSP algorithms (butterfly dag computation, cube dag computation, matrix multiplication). Other algorithms are obtained by application of established non-BSP techniques (sorting, randomised list contraction, Gaussian elimination without pivoting and with column pivoting, algebraic path computation), or use original techniques specific to the BSP model (deterministic list contraction, Gaussian elimination with nested block pivoting, communication-efficient multiplication of Boolean matrices, synchronisation-efficient shortest paths computation). The asymptotic BSP cost of each algorithm is established, along with its BSPRAM characteristics. We conclude by outlining some directions for future research.

Contents

Chapter 1

Introduction

The model of bulk-synchronous parallel (BSP) computation (see [Val90a, McC93, McC95, McC96c, McC98]) provides 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 cost analysis of communication and synchronisation.

Many other 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 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 the PRAM on more realistic models.

In contrast with the PRAM model, the BSP model accurately reflects the main design features of most existing parallel computers. On an abstract level, BSP is defined as a distributed memory model with point-to-point communication between the processors. Paper [Val90b] 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 thesis we propose a new model, called BSPRAM, which stands between BSP and PRAM. The BSPRAM model 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 the BSPRAM model. 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 technical computing. In the subsequent chapters we demonstrate the meaning and use of such properties by systematically designing algorithms for problems from three large, partially overlapping domains: combinatorial computation, dense matrix computation, graph computation. In view of our simulation results, BSPRAM here plays the role of a methodology for generic BSP algorithm design.

The algorithms presented in this thesis, as well as many other BSP algorithms, are defined for input sizes that are sufficiently large with respect to the number of processors. Apart from simplifying the BSPRAM algorithms, this condition provides the slackness and granularity necessary for their efficient BSP simulation. A typical form of such a condition is $n \geq poly(p)$, 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. 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 thesis 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", value n may not be an exact multiple of p, and therefore the blocks may differ in size by ± 1. Sometimes we use square bracket notation for matrices, referring to an element of an $n \times n$ matrix A as $A[i, j]$, $1 \leq i, j \leq n$. When the matrix is partitioned into regular blocks of size $m \times m$, we refer to each individual block as $A[[s, t]]$, $1 \leq s, t \leq n/m$.

Chapter 2

Bulk-synchronous parallel computation

2.1 Historical background

The last fifty years have seen the tremendous success of sequential computing. As pointed out in [Val90a, McC93, McC96c], 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. In 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 to the global state determined by the contents of the shared memory. In contrast, algorithms for distributed memory models have to be designed directly in terms of individual processors operating on their local memories.

- straightforward programming — the shared memory is uniformly accessible via a single address space using two basic primitives: reading

and writing. In 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. [CLR90, KR90, JáJ92, McC93]). 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 the PRAM model have been introduced. Among them are the *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 the *concurrent read, concurrent write PRAM (CRCW PRAM)*, which allows several processors to access a cell concurrently in one step. For the CRCW PRAM, a rule to resolve concurrent writing must be adopted. One of the possibilities, realised in the *combining CRCW PRAM* (see e.g. [CLR90, pages 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. [KR90, McC93]). A *circuit* is a directed acyclic graph (*dag*) with labeled nodes. We call a node *terminal*, if it is either a source (node of indegree 0), or a sink (node of outdegree 0). In a circuit, source nodes are labeled as *input*, sink nodes are labeled as *output*, and nonterminal nodes are labeled by arithmetic or Boolean operations. Algorithms that can be represented by 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 e.g. the standard $\Theta(n^3)$ method, or Strassen's $\Theta(n^{\log 7})$ method. In this thesis 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 [KRS90]). New parallel models were gradually introduced to account for resources other than the number of processors. Currently, dozens of such models exists; see [LMR96, MMT95, ST98] for a survey. Among the computer resources measured by these models are, according to [LMR96], 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 [Val90a, McC95, McC96c, McC98] 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 by 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. Such an extension is considered in [MT], where memory-efficient BSP algorithms for matrix multiplication are analysed.

In this thesis 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 and BSP by means of simulation. Let n denote the size of the input to a program. Following [Val90b], we say that a model A can *optimally simulate* a model B if there is a compilation algorithm that transforms any program with cost $T(n)$ 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.

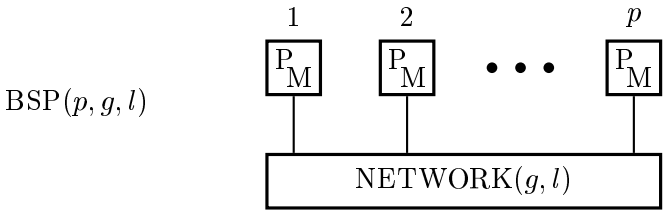


Figure 2.1: A BSP computer

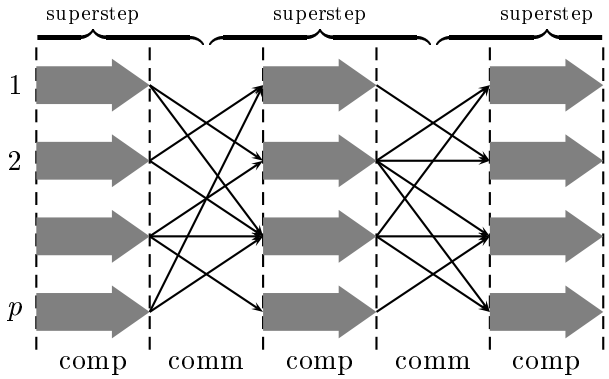


Figure 2.2: A BSP computation

2.2 The BSP model

A *BSP computer*, introduced in [Val89, Val90b, Val90a], consists of p *processors* connected by a communication network (see Figure 2.1). Each processor has a fast *local memory*. The processors may follow different threads of computation. A BSP computation is a sequence of *supersteps* (see Figure 2.2). 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.

Let *cost unit* be 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 $h = h' + h''$ (another possible definition is $h = \max(h', h'')$), then the cost of the superstep is defined as $w + h \cdot g + l$. Here g and l are the BSP parameters of the computer. The value g is the *communication throughput ratio* (also called "bandwidth inefficiency" or "gap"), the value l is the *communication latency* (also called "synchronisation periodicity").

We write BSP(p, g, l) to denote a BSP computer with the given values of p, g and l. If a computation consists of S supersteps with costs $w_s + h_s \cdot g + l$, $1 \leq s \leq S$, 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*. The values of W, H and S typically depend on the number of processors p and on the problem size. We define *the local computation volume* $\mathcal{W}$ as the total number of local operations, and *the communication volume* $\mathcal{H}$ as the total number of data units transferred between the processors. We call a BSP computation *balanced*, if $W = O(\mathcal{W}/p)$ and $H = O(\mathcal{H}/p)$.

In order to utilise the computer resources efficiently, a typical BSP program regards 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 $H = S = O(\log p)$ by a balanced binary tree, or with $H = O(p)$ and $S = O(1)$ by sending the value directly to every processor (this was observed in [Val90a]). On the other hand, a technique known as *two-phase broadcast* allows one to achieve perfect balance for the problem of broadcasting $n \geq p$ 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 $H = O(n)$ and $S = O(1)$ — both cost values are obviously optimal. Communication-optimal broadcasting of n values, $1 < n < p$, can be performed in $1 + \log p / \log n$ phases. The values are scattered so that each one gets to a distinct processor, then each value is broadcast by a balanced tree of degree n and height $\log p / \log n$. The communication and synchronisation costs of such simultaneous broadcast are $H = O(n \cdot \log p / \log n)$, $S = O(\log p / \log n)$. For $n = p^{\epsilon}$, where ϵ is a constant, $0 < \epsilon < 1$, both cost values are trivially optimal. For any asymptotically smaller n, there is a communication-synchronisation tradeoff.

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.

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 the *automatic mode* of BSP programming, as opposed to the *direct mode*, i.e. programming with explicit

control over memory management.

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 BSP computer with 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. 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 [Val90b] provides the following result.

Theorem 1. *Let $g = O(1)$, $l = O(\sigma)$. An optimal randomised simulation on BSP(p, g, l) can be achieved for*

(*i*) *any EREW PRAM algorithm with slackness $\sigma \geq \log p$;*

(*ii*) *any CRCW PRAM algorithm with slackness $\sigma \geq p^{\epsilon}$ for a constant $\epsilon > 0$.*

Proof. See [Val90b]. ■

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 local memory. This lack of data locality may be insignificant for highly irregular problems (e.g. multiplication of sparse matrices with a random pattern of nonzeros). On the other hand, data locality should be preserved when dealing with more structured problems (e.g. multiplication of dense matrices, or sparse matrices with a regular nonzero pattern). Efficient BSP solution of such problems cannot be achieved via 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

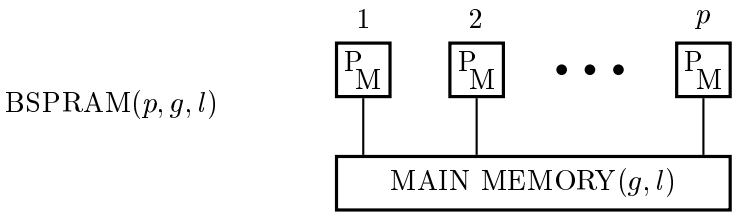


Figure 2.3: A BSPRAM

a suitably adapted concept of slackness. We also describe a deterministic simulation, based on additional properties of obliviousness and granularity.

2.3 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 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 the PRAM simulation method 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 Figure 2.3). In addition, there is a single shared *main memory*. As in BSP, the computation proceeds by *supersteps* (see Figure 2.4). 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.

As with the PRAM, concurrent access to the main memory in one superstep can be either allowed or disallowed. In this thesis 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 mem-

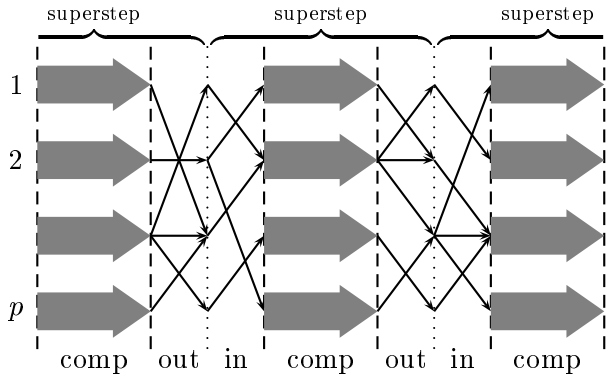


Figure 2.4: A BSPRAM computation

ory. 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. [CLR90]).

In a similar way to the BSP model, the cost of a BSPRAM superstep is defined 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 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 [ACS90]. 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 [ACS90] is CREW. The model has an explicit bandwidth parameter, corresponding 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 the Asynchronous PRAM, was proposed in [Gib93] (an earlier version of this model was called the Phase

PRAM). Like BSPRAM, the 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 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 [GMR99, GMR, Ram99] (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$(p, g, 1)$.

An interesting partial alternative to shared memory bulk-synchronous parallel models is offered by array languages with implicit parallelism. An example of such a language is ZPL (see [Sny98]), based on the CTA/Phase Abstractions model described in [AGL+98]. The developers of ZPL have announced their plans for an extension, called Advanced ZPL, which is likely to be similar to the BSPRAM model.

Just as for PRAM simulation, some "extra parallelism" is necessary for efficient BSPRAM simulation on BSP. We say that a BSP or 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 $\sigma \geq \log p$;*

(*ii*) *any CRCW BSPRAM(p, g, l) algorithm with slackness $\sigma \geq p^{\epsilon}$ for a constant $\epsilon > 0$.*

Proof. Immediately follows from Theorem 1. ■

In contrast with Theorem 1, no conditions on g and l are necessary, since the simulated and the simulating machines have the same BSP parameters.

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. [KR90]). We say that a BSPRAM algorithm is *communication-oblivious*, if the sequence of communication and synchronisation operations executed by a processor is the same for any input of a given size (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. The slackness of a BSPRAM algorithm will always be at least as large as its granularity: $\sigma \geq \gamma$.

Communication-oblivious BSPRAM algorithms, and BSPRAM algorithms with sufficient granularity, allow optimal deterministic BSP simulation. 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 $(1+\epsilon^{-1})$-phase broadcast described in the previous section.

Below we formally state the results on deterministic BSPRAM simulation, published previously in [Tis96, Tis98].

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 $\sigma \geq p^\epsilon$ for a constant $\epsilon > 0$;*

(*iii*) *any CRCW BSPRAM(p,g,l) algorithm with granularity $\gamma \geq p$.*

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 [Tis96]. Its difference from direct BSP mode is 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 known (see e.g. [Ber85, page 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 simultaneous broadcast technique from Section 2.2. Without loss of generality, we assume that the communication cost of the considered input phase is $h = \sigma = p^\epsilon$, $0 < \epsilon < 1$. 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 $2\epsilon^{-1}$, 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. ■

The proofs of Theorems 2 and 3 show that a BSP computer can execute many practical BSPRAM algorithms within a low constant factor of their cost. For two important classes of algorithms — communication-oblivious algorithms and algorithms with sufficient granularity — the simulation is deterministic and particularly simple.

It is intuitively clear that in general, the BSPRAM shared memory mechanism is at least as powerful as BSP message passing. However, not every BSP algorithm can be optimally simulated on a BSPRAM, due to different input-output conventions. The following result gives a simulation, which is sufficient for most practical applications.

Theorem 4. *An optimal deterministic simulation on an EREW BSPRAM*(p, g, l) *can be achieved for any BSP*(p, g, l) *algorithm with slackness* $\sigma \geq p$.

Proof. The main memory of a BSPRAM is partitioned into p^2 areas. Each area corresponds to a pair of communicating BSP processors. Sending a message from processor p_1 to processor p_2 is implemented by p_1 writing the message, preceded by its length, to the area corresponding to the pair p, q. Receiving this message is implemented by q reading the length, and then the message (if the length is nonzero). ■

In the rest of this thesis, we develop and analyse BSP algorithms for some common computational problems. The BSPRAM model is used as the

basis of our presentation. Sometimes it is convenient to use BSP message passing for (some part of) the computation, while keeping BSPRAM shared memory for input-output. In such cases we extend the BSPRAM model by assuming that the processors, in addition to the shared memory, are connected by a BSP-style communication network. In this extended model, a computation is a mixture of BSPRAM-style and BSP-style supersteps; we will say that the computation switches between *shared-memory mode* and *message-passing mode*. Such mixed algorithms can be translated into pure BSP by the simulation mechanisms of Theorems 2, 3, and into pure BSPRAM by Theorem 4.

When analysing slackness and granularity, we will often ignore a part of the computation which is non-critical, i.e. does not affect the asymptotic cost of the whole algorithm. The reason for it is that such non-critical computation may be simulated non-optimally without reducing the overall performance. Every time when such an omission is made, we will indicate explicitly the part of the computation which is considered non-critical.

Chapter 3

Combinatorial computation in the BSP model

3.1 Complete binary tree computation

One of the main divisions in computer science is between "algebraic" and "combinatorial" computation. Both terms are usually understood in a broad sense, and some algorithms fall under both categories. We will consider computational problems of an algebraic nature in Chapters 4 and 5. In this chapter, we concentrate on simple combinatorial objects, such as dags, arrays, linked lists and trees. The first few sections deal with BSP computation of dags.

As defined in Section 2.1, a dag is a directed acyclic graph. We will usually ignore terminal nodes when establishing the size and the depth of a dag. Also, we will not show terminal nodes in pictures of dags, when this does not create confusion. By computation of a dag we understand computation of a circuit based on that dag. When a dag is computed in parallel, we call an edge $u \to v$ *local*, if it does not require communication — that is, the set of processors computing v is a subset of the set of processors computing u. We call a node v *local*, if all its incoming edges are local — that is, any processor computing v computes also all predecessors of v.

The communication cost of parallel dag computation has been analysed e.g. in [PU87, PY90, JKS93]. These papers adopt a synchronous communication cost model, where a nonlocal edge incurs a fixed *communication delay*. The number of processors is unbounded. A node may be computed, in general, more than once by different processors. Paper [JKS93] shows that such recomputation of nodes is necessary for an asymptotically optimal computation of certain dags in the given model.

In a BSP dag computation, we also allow recomputation of nodes. However, it is not required by any of the algorithms in this chapter.

We begin with a simple problem of computing a circuit based on a com-

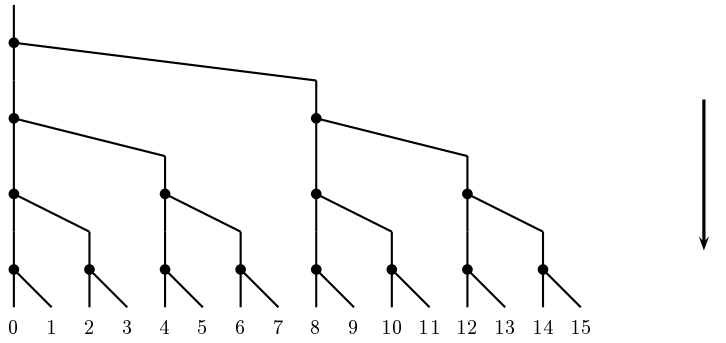


Figure 3.1: Complete binary tree $cbtree(16)$

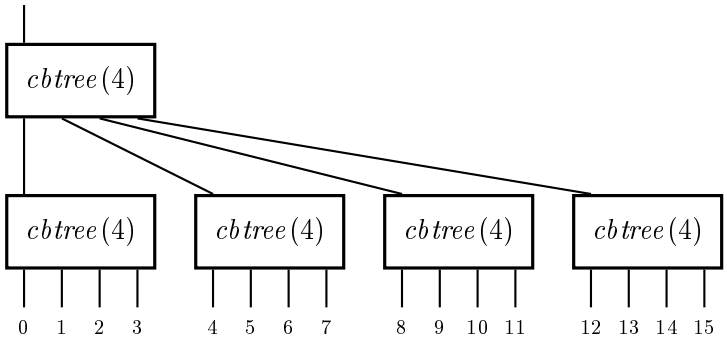


Figure 3.2: BSPRAM computation of $cbtree(16)$

plete binary tree. This problem often occurs in practice, and has important applications to other problems, such as computing all-prefix sums. The analysis of complete binary tree computation will help us to illustrate the BSPRAM model, as well as the concepts of communication-obliviousness, slackness and granularity.

As in any circuit, each node in the tree represents an elementary operation. Depending on the nature of the computation, we can view the node's parent as the input and the children as the outputs, or vice versa. We will call these two versions of tree computation *top-to-bottom* and *bottom-to-top*, respectively. By symmetry, we need to analyse only one of the two. We choose top-to-bottom computation, which generalises the broadcast problem considered in Section 2.2 (the operation of each node in the case of broadcast is simple replication of the input). We denote the complete one-input, n-output binary tree dag by $cbtree(n)$. This dag has $n-1$ nonterminal nodes, and depth $\log n$. Figure 3.1 shows the dag $cbtree(16)$.

For a BSPRAM computation of the tree, we assume that the input and the outputs are stored in the main memory. The computation method resembles the broadcast techniques from Section 2.2. Figure 3.2 shows the computation for $n = 16$, $p = 4$.

Algorithm 1. *Computation of a complete binary tree.*

Parameters: integer $n \geq p^{1+\epsilon}$ for some constant $\epsilon > 0$; a circuit based on $cbtree(n)$.

Input: value x.

Output: values y_i, $0 \leq i < n$, computed by the circuit.

Description. The computation is performed on an EREW BSPRAM(p, g, l) in shared-memory or message-passing mode. If $\epsilon \geq 1$, the computation proceeds in two supersteps. The first superstep computes the top $\log p$ levels of the tree, and the second superstep the remaining $\log n - \log p$ levels. If $0 < \epsilon < 1$, the computation proceeds in $1 + \epsilon^{-1}$ supersteps. Each superstep computes $\epsilon \log p$ levels of the tree, except the last superstep, which computes the remaining $\log n - \log p$ levels.

Cost analysis. For $\epsilon \geq 1$, the local computation, communication and synchronisation costs are

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

For $0 < \epsilon < 1$, the local computation, communication and synchronisation costs are

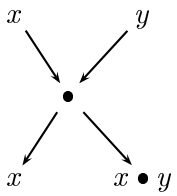
$$W = O(n/p) \qquad H = O(\epsilon^{-1} \cdot n/p) = O(n/p) \qquad S = O(\epsilon^{-1}) = O(1)$$

The algorithm is oblivious, with slackness and granularity $\sigma = \gamma = 1$. ■

Bottom-to-top complete binary tree computation is symmetric to the algorithm above.

In Algorithm 1, each of the three cost values W, H, S, taken independently, is trivially optimal.

An important application of complete binary trees is the problem of computing all-prefix sums on an array of size n (see e.g. [Ble93, LD94]). The problem is formulated as follows: given an input array $(x_0, x_1, \ldots, x_{n-1})$, compute the output $(y_0, y_1, \ldots, y_{n-1}) = (x_0, x_0 \bullet x_1, \ldots, x_0 \bullet x_1 \bullet \cdots \bullet x_{n-1})$, where $\bullet$ is an associative operator computable in time $O(1)$. A standard method of computing all-prefix sums in parallel, proposed in [BK82] (see also [LD94]), can be represented by a dag $allpref(n)$, shown in Figure 3.3 for $n = 8$. Here, the action of a node with inputs x, y is



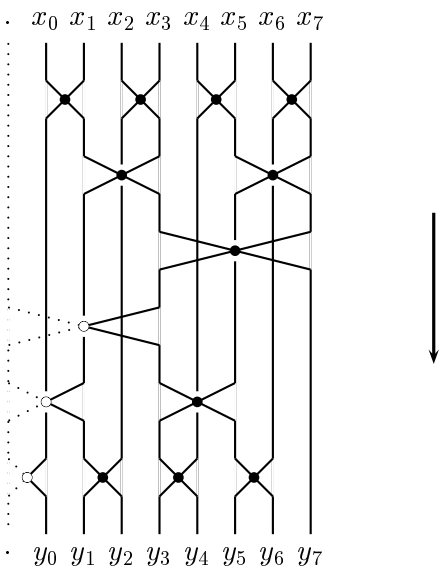
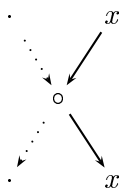


Figure 3.3: All-prefix sums dag *allpref*(8)

To expose the symmetry of the method, we have introduced a dummy value, shown in Figure 3.3 by a dot $\cdot$. The action of a node with inputs $\cdot$, x is



The dag *allpref*(n) consists of two linked complete binary trees, the first computed bottom-to-top, the second top-to-bottom. In total, the dag *allpref*(n) has $2n-2$ nonterminal nodes, and depth $2\log n$. If $n \geq p^{1+\epsilon}$ for a constant $\epsilon > 0$, Algorithm 1 allows one to compute all-prefix sums with BSP cost $W = O(n/p)$, $H = O(n/p)$, $S = O(1)$.

3.2 Butterfly dag computation

The butterfly dag represents the dependence pattern of the Fast Fourier Transform. Another application of the butterfly dag is in the bitonic sorting network (see e.g. [CLR90]). Parallel algorithms for the butterfly dag computation have been proposed in various parallel models (see e.g. [CLR90, JáJ92]).

The *butterfly dag bfly*(n) takes n inputs x_i, and produces n outputs y_i, $0 \leq i < n$. The dag contains $\log n$ levels of nodes, with $n/2$ nodes in each level. For all i, $0 \leq i < n$, let us define $u_i^0 = x_i$, and let u_i^k, $1 \leq k \leq \log n$,

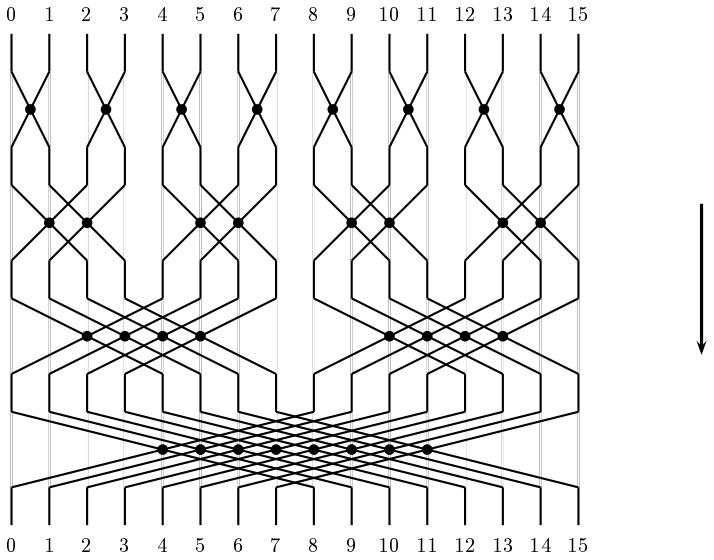


Figure 3.4: Butterfly dag *bfly*(16)

denote the output of level $k-1$, so that $u_i^{\log n} = y_i$. In level k, there is a node with inputs u_i^k, u_j^k, and with outputs u_i^{k+1}, u_j^{k+1}, if and only if $|j-i| = 2^k$. In total, there are $1/2 \cdot n \log n$ nonterminal nodes, and the depth of the dag is $\log n$. Figure 3.4 shows the butterfly dag *bfly*(16).

As observed in [PY90, Val90a] (see also [GHSJ96]), the butterfly dag can be partitioned in a way suitable for bulk-synchronous parallel computation. The computation of a level in *bfly*(n) consists of $n/2$ independent computations of *bfly*(2). Similarly, the computation of any k consecutive levels consists of $n/2^k$ independent computations of *bfly*(2^k). Therefore, the butterfly dag computation can be split into two stages, each comprising $1/2 \cdot \log n$ 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.

Figure 3.5 shows the two-superstep computation of *bfly*(16). In each superstep, four independent computations of *bfly*(4) are performed. In general, the algorithm is as follows.

Algorithm 2. *Computation of the butterfly dag bfly*(n).

Parameters: integer $n \geq p^2$; a circuit based on *bfly*(n).

Input: values x_i, $0 \leq i < n$.

Output: values y_i, $0 \leq i < n$, computed by the circuit.

Description. The computation is performed on an EREW BSPRAM(p, g, l) and proceeds in two supersteps, each comprising $1/2 \cdot \log n$ levels. In both

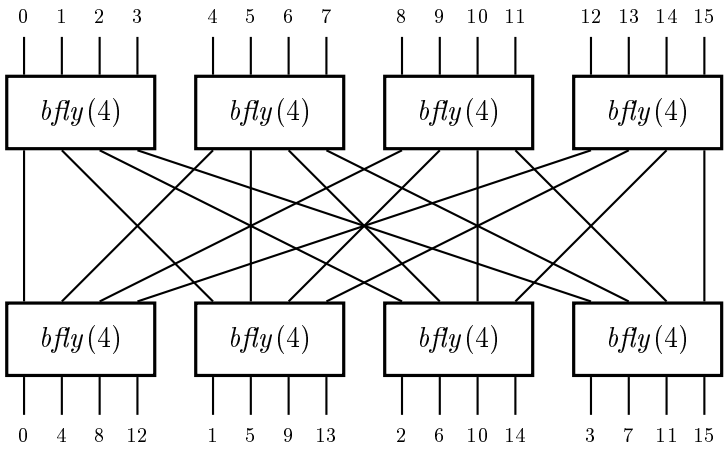


Figure 3.5: BSPRAM computation of $bfly(16)$

supersteps, each processor is assigned $n^{1/2}/p$ independent butterfly dags of size $n^{1/2}$.

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 $\gamma = n/p^2$. ■

The asymptotic BSP costs of Algorithm 2 are independently optimal. Efficient BSP computation of a butterfly dag for $1 \leq n < p^2$ is considered in [Val90a].

3.3 Cube dag computation

The cube dag defines the dependence pattern that characterises a large number of scientific algorithms. Here we describe a BSPRAM version of the BSP cube dag algorithm from [McC95]. 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 $y^{(1)}_{jk}$, $y^{(2)}_{ik}$, $y^{(3)}_{ij}$, $0 \leq i, j, k < n$, contains n^3 nonterminal nodes v_{ijk}, such that

$$\begin{aligned}
&v_{0jk},\ v_{i0k},\ v_{ij0} \text{ input respectively } x^{(1)}_{jk},\ x^{(2)}_{ik},\ x^{(3)}_{ij} \\
&v_{ijk} \text{ is connected to each of the nodes} \\
&\qquad v_{i+1,j,k},\ v_{i,j+1,k},\ v_{i,j,k+1} \text{ whenever such node exists} \\
&v_{n-1,j,k},\ v_{i,n-1,k},\ v_{i,j,n-1} \text{ produce respectively } y^{(1)}_{jk},\ y^{(2)}_{ik},\ y^{(3)}_{ij}
\end{aligned} \tag{3.1}$$

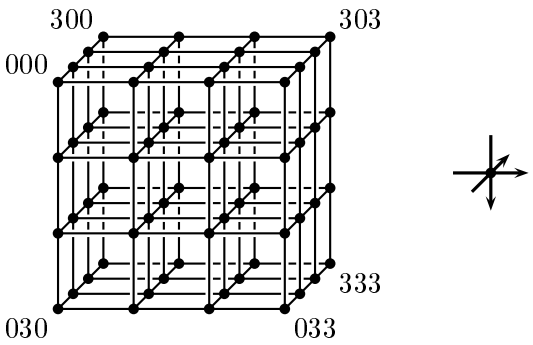


Figure 3.6: Cube dag $cube_3(4)$

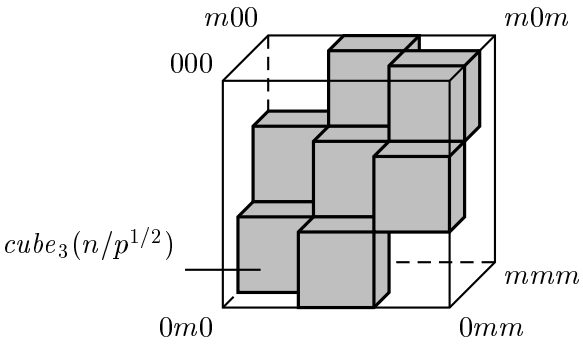


Figure 3.7: BSPRAM computation of $cube_3(n)$, $m = n - 1$

The depth of the dag is $3n - 2$. Figure 3.6 shows the cube dag $cube_3(4)$.

The BSP algorithm for computing the dag $cube_3(n)$ is given in [McC95]. In this algorithm, the array $V = (v_{ijk})$ is partitioned into $p^{3/2}$ regular cubic blocks of size $n/p^{1/2}$. We denote these blocks by V_{ijk}, $0 \le i, j, k < p^{1/2}$. Each block defines a dag isomorphic to $cube_3\big(n/p^{1/2}\big)$. The algorithm computes a block V_{ijk} as soon as the data from its predecessors $V_{i-1,j,k}$, $V_{i,j-1,k}$, $V_{i,j,k-1}$ become available. The diagonal layer of simultaneously computed independent blocks forms the computation "wavefront".

Figure 3.7 shows a stage in the BSP computation of $cube_3(n)$. The shaded diagonal layer of blocks is the current wavefront. The total number of layers is $3p^{1/2} - 2$, therefore the computation can be completed in $O(p^{1/2})$ supersteps.

Algorithm 3. *Computation of the cube dag* $cube_3(n)$.

Parameters: integer $n \ge p^{1/2}$; a circuit based on $cube_3(n)$.

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

Output: values $y^{(1)}_{jk}$, $y^{(2)}_{ik}$, $y^{(3)}_{ij}$, $0 \le i, j, k < n$, computed by the circuit.

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

Cost analysis. The local computation cost is $W = O(n^3/p)$. The computation of a block requires the communication 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 $H = h \cdot p^{1/2} = O\big(n^2/p^{1/2}\big)$. The synchronisation cost is $S = O\big(p^{1/2}\big)$. The algorithm is oblivious, with slackness and granularity $\sigma = \gamma = n^2/p$. ■

The definition of a three-dimensional cube dag can be naturally generalised to any dimension. A two-dimensional dag is called the *diamond dag*. It can be computed by an algorithm similar to Algorithm 3 with BSP cost $W = O(n^2/p)$, $H = O(n)$, $S = O(p)$. In general, the BSP cost of computing a cube dag $cube_d(n)$ is $W = O(n^d/p)$, $H = O\big(n^{d-1}/p^{\frac{d-2}{d-1}}\big)$, $S = O\big(p^{\frac{1}{d-1}}\big)$.

The BSP cost values of Algorithm 3 are not independently optimal. However, H and S are optimal for any computation with $W = O(n^3/p)$. To prove the optimality in H, we need the following lemma.

Lemma 1. *Any BSPRAM(p, g, l) computation of $cube_3(n)$ with local computation cost $W \leq 5/36 \cdot n^3$ requires communication volume $\mathcal{H} \geq 1/6 \cdot n^2$.*

Proof. Suppose the communication volume is less than $1/6 \cdot n^2$. Then the dag must have less than $1/6 \cdot n^2$ nonlocal nodes. Partition the dag into n parallel planes. The middle plane divides the dag into a lower-indexed and a higher-indexed half. At least one of the planes in the higher-indexed half contains less than $(1/6 \cdot n^2)/(1/2 \cdot n) = 1/3 \cdot n$ nonlocal nodes; we will call it *the base plane*. Consider two orthogonal partitionings of the base plane into n parallel lines. In each partitioning, there are more than $2/3 \cdot n$ lines consisting of local nodes only. The intersection of these two line families contains more than $(2/3 \cdot n)^2 = 4/9 \cdot n^2$ nodes. We call this intersection *the base diamond*, and the highest-indexed node in the base diamond *the base node*.

Consider the set of lines intersecting the base plane orthogonally at the base diamond. More than $4/9 \cdot n^2 - 1/6 \cdot n^2 = 5/18 \cdot n^2$ of the lines consist of local nodes only. In total, we have more than $5/18 \cdot n^3$ local nodes, $5/36 \cdot n^3$ of which are in the lower-indexed half of the dag. By construction, each of these nodes must be computed by at least the same processors as the base node. Therefore, the local computation cost is more than $5/36 \cdot n^3$. Conversely, if the local computation cost is at most $5/36 \cdot n^3$, the communication volume is at least $1/6 \cdot n^2$. ■

The conditional optimality of Algorithm 3 can now be demonstrated as follows.

Theorem 5. *Any BSPRAM(p, g, l) computation of cube$_3(n)$ with $W = O(n^3/p)$ requires (i) $W = \Theta(n^3/p)$, (ii) $H = \Omega\left(n^2/p^{1/2}\right)$, (iii) $S = \Omega\left(p^{1/2}\right)$.*

Proof. (i) Trivial.

(ii) The proof is an extension of the proof given in [PU87] for the diamond dag.

Let $W = O(n^3/p)$. Partition the cube dag into $p^{3/2}$ cubic blocks of size $n/p^{1/2}$. Consider $3p$ chains of blocks parallel to the main diagonal. In every chain, the computation of a block can start only after each node in the previous block has been computed at least once. For the purpose of a lower bound, we will ignore all computation in a block after the highest-indexed node has been computed once.

There are $3/4 \cdot p$ "long" chains, each containing at least $1/2 \cdot p^{1/2}$ blocks. Since the local computation cost of each chain is $O(n^3/p)$, there are $\Omega\left(p^{1/2}\right)$ blocks in each "long" chain with local computation cost at most $5/36 \cdot n^3/p^{3/2}$. By Lemma 1, the communication volume of such a block is at least $1/6 \cdot n^2/p$. The total number of such blocks is $(3/4 \cdot p) \cdot \Omega\left(p^{1/2}\right) = \Omega\left(p^{3/2}\right)$. Therefore, the total communication volume must be at least $(1/6 \cdot n^2/p) \cdot \Omega\left(p^{3/2}\right) = \Omega\left(n^2 \cdot p^{1/2}\right)$. Even when communication is perfectly balanced, the total communication cost $H = \Omega\left(n^2 \cdot p^{1/2}\right)/p = \Omega\left(n^2/p^{1/2}\right)$.

(iii) Nodes of the main diagonal v_{iii}, $0 \le i < n$, first computed in each particular superstep, form a consecutive segment. We denote the sizes of these segments by m_s, $0 \le s < S$, where $\sum_{0 \le s < S} m_s = n$. Since the communication within a superstep is not allowed, all nodes in the diagonal cubic block of size m_s spanned by the segment s must be computed by the processor that first computes the highest-indexed node of the block. The local computation cost of a block is therefore m_s^3. The total cost of local computation is bounded from below by the sum over all supersteps: $\sum_{0 \le s < S} m_s^3$. By Hölder's inequality,

$$n = \sum_{0 \le s < S} m_s = \sum_{0 \le s < S} 1 \cdot m_s \le S^{2/3} \cdot \left(\sum_{0 \le s < S} m_s^3 \right)^{1/3} \le S^{2/3} \cdot W^{1/3}$$

Hence by assumption $S \ge n^{3/2}/W^{1/2} = \Omega(p^{1/2})$. ■

Theorem 5 can be easily generalised to other dimensions.

3.4 Sorting

Sorting is a classical problem of parallel computing. Many parallel sorting algorithms of different complexity have been proposed (see e.g. [GR88, JáJ92, Col93, TB95] and references therein). Here we consider comparison-based sorting of an array $\mathbf{x} = (x_i)$, $1 \le i \le n$. Without loss of generality,

we may assume that the elements of $\mathbf{x}$ 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 $\mathbf{x}$ such that $a < x < b$.

Probably the simplest parallel sorting algorithm is parallel sorting by regular sampling (PSRS), proposed in [SS92] (see also its discussion in [LLS+93]). Paper [HJB] 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 $\mathbf{x}$ is partitioned into p subarrays $\mathbf{x}^1, \ldots, \mathbf{x}^p$, each of size n/p. The subarrays $\mathbf{x}^q$ are sorted independently by an optimal sequential algorithm. The problem now consists in merging the p sorted subarrays.

In 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 included among the samples). We denote the samples of the subarray $\mathbf{x}^q$ by $\bar{x}^q_0, \ldots, \bar{x}^q_p$. The samples divide each subarray into p *primary blocks* of size at most n/p^2. We denote the primary blocks of $\mathbf{x}^q$ by $\left[\bar{x}^q_0, \bar{x}^q_1\right], \ldots, \left[\bar{x}^q_{p-1}, \bar{x}^q_p\right]$. Then, $p \cdot (p+1)$ primary samples are collected together and sorted by an arbitrary sequential algorithm. After that, we select $p+1$ 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 $\mathbf{x}$ into p *secondary blocks*, corresponding to the intervals $\left\langle \bar{\bar{x}}_0, \bar{\bar{x}}_1 \right\rangle, \ldots, \left\langle \bar{\bar{x}}_{p-1}, \bar{\bar{x}}_p \right\rangle$. Each secondary block is distributed across the processors. Now it remains to collect the elements of each secondary block in one particular processor.

Let us show that any secondary block contains at most $3n/p$ elements. For a fixed secondary block defined by $\left\langle \bar{\bar{x}}_k, \bar{\bar{x}}_{k+1} \right\rangle$, we divide all the primary blocks of $\mathbf{x}$ into three categories. We call a primary block $\left[\bar{x}^q_i, \bar{x}^q_{i+1}\right]$ an *inner block*, if $\left\langle \bar{x}^q_i, \bar{x}^q_{i+1} \right\rangle \subseteq \left\langle \bar{\bar{x}}_k, \bar{\bar{x}}_{k+1} \right\rangle$; an *outer block*, if $\left\langle \bar{x}^q_i, \bar{x}^q_{i+1} \right\rangle \cap \left\langle \bar{\bar{x}}_k, \bar{\bar{x}}_{k+1} \right\rangle = \emptyset$; 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 one or both secondary block boundaries). Therefore, the size of a secondary block is at most $n/p^2 \cdot (p + 2p) = 3n/p$. In the second stage of merging, the elements of each secondary block can be collected in time $O(n/p)$, and then sorted by an efficient sequential algorithm.

The method is illustrated in Figure 3.8 for $p = 3$. The state of the array $\mathbf{x}$ 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; note that the order of primary samples from each subarray is preserved, but the samples from different subarrays may be interleaved. The dashed bars at the bottom show the elements of $\mathbf{x}$ assumed to lie between the samples;

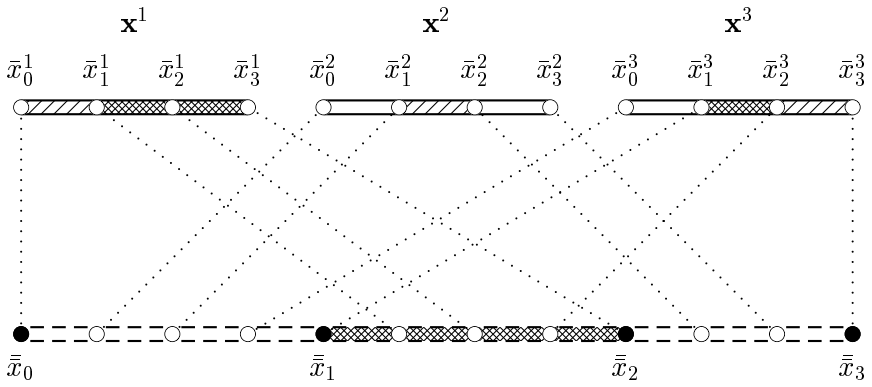


Figure 3.8: BSPRAM sorting by regular sampling

the number of elements between adjacent primary samples is not necessarily equal. Black dots indicate the secondary samples. The secondary block $\langle \bar{\bar{x}}_1, \bar{\bar{x}}_2 \rangle$ is shown by cross-hatching. Primary blocks that are inner, boundary and outer for $\langle \bar{\bar{x}}_1, \bar{\bar{x}}_2 \rangle$ are shown by cross-hatching, simple hatching and no hatching 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.

Algorithm 4. *Sorting by regular sampling.*

Parameter: integer $n \geq p^3$.

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

Output: $\mathbf{x}$ rearranged in increasing order.

Description. The computation is performed on a CRCW BSPRAM(p, g, l) and proceeds in three supersteps. In the first superstep, a processor picks a subarray $\mathbf{x}^q$, reads it, sorts it with an efficient sequential algorithm, selects $p+1$ 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. Efficiency of the above computation with samples is not critical, since the number of samples does not depend on n. 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 (in message-passing mode) 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 received 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

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

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

Lower bounds on communication complexity of sorting for various parallel models can be found e.g. in [SS92, ABK95]. The asymptotic BSP costs of Algorithm 4 are independently optimal.

Paper [Goo96] presents a more complex BSP sorting algorithm, asymptotically optimal for any $n \geq p$. Its BSP costs are $W = O(n \log n/p)$, $H = O\big(n/p \cdot \log n/\log(n/p)\big)$, $S = O\big(\log n/\log(n/p)\big)$. For $n \geq p^3$, 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. [Col93]). Despite its asymptotic optimality, the algorithm from [Goo96] 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 [GS96].

3.5 List contraction

This and the following sections consider BSPRAM computation on pointer structures, such as linked lists and trees. A *linked list* is a sequence of *items*. The order of items is defined by pointers: each item contains a pointer to the next item in the sequence. The last item contains the null pointer. The first and the last items of the list are called its *head* and *tail*, respectively. A *doubly-linked list* contains backward as well as forward pointers.

The most common problem on linked (or doubly-linked) lists is *list ranking*: for each item determine its distance from the head (or the tail) of the list (see e.g. [CLR90, JáJ92, RMMM93]). List ranking can be applied to more general list problems, such as computing all-prefix sums on a list. Following [LM88], we view these problems as instances of an abstract problem of *list contraction*: given an abstract operation of *merging* two adjacent items as a primitive, contract the list to a single item. Implementation of the merging primitive is problem-dependent; it usually involves pointer jumping and some payload operations (e.g. summation of item values). We assume that the computation cost of merging two items is $O(1)$.

In a sequential model of computation, a list can be contracted by a trivial algorithm that traverses the list of n items in $\Theta(n)$ time. The problem is rather more complicated on parallel models. The easiest way to obtain

an efficient parallel list contraction algorithm is by randomisation. Paper [MR85] introduced a technique of *random mating*. The random mating algorithm proceeds in a sequence of rounds. In each round every item is marked either *forward-looking* of *backward-looking* by flipping an independent unbiased coin. Then pairs of adjacent items that "look at each other" merge. The procedure is repeated until only one item is left. One round of the algorithm reduces the size of the list by about a quarter, therefore the expected number of parallel steps is $\Theta(\log n)$.

The expected amount of computation performed by the above algorithm is optimal; however, in the PRAM model the time-processor product[1] is still suboptimal. Many attempts have been made to improve the PRAM time-processor efficiency of randomised list contraction. An algorithm from [RM96] is time-processor optimal. Although it is slightly suboptimal in time, it performs better in practice than the more sophisticated algorithm from [AM90], optimal both in time and in the time-processor product.

Optimal efficiency for randomised list contraction is much easier to achieve in the BSPRAM model, given sufficient slackness. The following straightforward implementation of random mating is based on a BSP algorithm suggested by [McC96a].

Algorithm 5. *Randomised list contraction.*

Parameter: integer $n \geq p^2 \cdot \log p$.

Input: linked list of size n.

Output: input list contracted to a single item.

Description. The computation is performed on an EREW BSPRAM(p, g, l). Each processor reads an equal number of input items from the main memory. After that, the computation is performed in message-passing mode and proceeds in two stages.

First stage. We reduce the list from n to n/p items by repeated rounds of random mating. Each round is implemented by a superstep, and consists in merging all mating pairs. The processor to hold each merged pair is chosen at random (other methods of choice are possible).

Second stage. The remaining n/p items are collected in a single processor. This processor completes the contraction by local computation.

Cost analysis. An analysis along the lines of [LM88] shows that $O(\log p)$ rounds will suffice on the first stage with high probability. Since the size of the list is expected to decrease exponentially, the communication cost of

[1] This time-processor product is also sometimes called 'work'; we use the term work for the actual number of operations performed, which may be smaller than the time-processor product, due to some processors being idle for some time.

the first round of mating, equal to $O(n/p)$, dominates all subsequent communication with high probability. Expected (with high probability) local computation, communication and synchronisation costs of the whole algorithm are

$$W_{\exp} = O(n/p) \qquad H_{\exp} = O(n/p) \qquad S_{\exp} = O(\log p)$$

The algorithm is not communication-oblivious. Its expected slackness is $\sigma_{\exp} = n/p^2$. Its granularity is $\gamma = 1$. ■

Another direction of research has been aimed at providing an optimal deterministic algorithm for list contraction. Known efficient deterministic algorithms for PRAM (see e.g. [JáJ92, RMMM93]) typically involve the method of symmetry breaking by deterministic coin tossing introduced in [CV86]. Such algorithms are complicated and often assume non-standard arithmetic capabilities of the computational model, e.g. bitwise operations on integers. As in the case of randomised algorithms, it is much easier to design an optimal deterministic algorithm for list contraction in the BSP model, provided that the input size is sufficient. Our deterministic list contraction algorithm is based on the technique of *deterministic mating*, described below.

The algorithm proceeds in several rounds. Each round starts with contracting all chains of adjacent items that are local to any particular processor. Any item that remains in the list has both neighbours outside its containing processor.

After that, a complete weighted digraph is constructed. The graph has p nodes, each node representing a processor. The weight of an edge $v_1 \to v_2$ is defined as the number of adjacent pairs of items, where the leading and the trailing item are contained in the processor represented by v_1 and v_2 respectively. The graph is used to mark each processor either *forward-looking* or *backward-looking*. Let m be the total number of items before the current round. The forward and backward marks are assigned in such a way that the number of adjacent pairs of items "looking at each other" is at least $m/4$. Such a marking always exists and can be easily computed from the graph by a greedy algorithm in sequential time $O(p^2)$.

Each item assumes the mark of the containing processor. Then pairs of neighbours that "look at each other" merge. At this stage, the total number of remaining items is at most $3m/4$, but their distribution across the processors may not be even. Therefore, it is necessary to redistribute the items so that each processor receives at most $3m/4p$ of them. This completes the current round.

The BSPRAM implementation of deterministic mating is as follows.

Algorithm 6. *Deterministic list contraction.*

Parameter: integer $n \geq p^3 \cdot \log p$.

Input: linked list of size n.

Output: input list contracted to a single item.

Description. The computation is performed on an EREW BSPRAM(p, g, l). Each processor reads an equal number of input items from the main memory. After that, the computation is performed in message-passing mode and proceeds in two stages.

First stage. We reduce the list from n to n/p items by repeated rounds of deterministic mating. Each round is implemented by three supersteps.

In the first superstep, each processor reduces all local chains and computes the number of links from local items to items in each of the other processors. This establishes the weights of edges leaving the processor's node in the representing graph. After that, the whole graph is collected in a single processor. This processor computes the marks and tells each processor its mark.

In the second superstep, pairs of adjacent items that "look at each other" merge. The processor to hold the merged pair is chosen arbitrarily between the two processors holding the original items.

The third superstep redistributes the items so that each processor receives at most $3m/4p$ of them. This completes the current round.

Second stage. The remaining n/p items are collected in a single processor. This processor completes the contraction by local computation.

Cost analysis. The total number of rounds necessary to reduce the list to n/p items in the first stage is $O(\log p)$. Since the size of the list decreases exponentially, the communication cost of the first round, equal to $O(n/p)$, dominates all subsequent communication. The local computation, communication and synchronisation costs of the whole algorithm are

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

The algorithm is not communication-oblivious. Its slackness and granularity are $\sigma = n/p^2$, $\gamma = 1$ (ignoring non-critical computation of processor marks). ■

In Algorithms 5 and 6, the asymptotic values of W, H, S are not independently optimal. Intuitively, it seems likely that H and S are optimal for any computation with $W = O(n/p)$, but the question of a proof remains open. Some lower bounds for parallel list contraction have been proved in [Sib97].

3.6 Tree contraction

Both the randomised and the deterministic versions of list contraction can be used to solve the problem of *tree contraction*. This well-studied problem (see

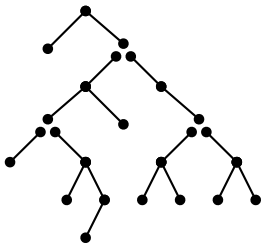


Figure 3.9: BSPRAM tree contraction

e.g. [JáJ92, RMMM93]) generalises list contraction. As before, the problem is defined in terms of an abstract operation of *merging* two adjacent nodes in a binary tree; the merging operation is considered primitive. In tree contraction, two kinds of merging are allowed:

- *raking*, where a leaf is absorbed into its parent node. The parent and child of the resulting node are respectively the parent and the remaining child of the absorbing node.

- *compression*, where a non-leaf with only one child absorbs its child. The parent and children of the resulting node are respectively the parent of and the absorbing node and the children of the absorbed node.

When the absorbing node has only one child, which is a leaf, merging can be classified both as raking and compression.

Similarly to list contraction, the goal of tree contraction is to reduce the tree to a single node. Tree contraction provides an efficient solution to problems connected with parallel evaluation of arithmetic expressions. It is also used as a subroutine in some parallel graph algorithms, such as the minimum spanning tree computation.

Several approaches to tree contraction have been developed for the PRAM model. One method to obtain an efficient PRAM algorithm for tree contraction is by generalising the technique of random mating (see e.g. [LM88]). Another possibility is to reduce the problem to list contraction by considering lists associated with the tree, such as its Euler tour. The latter approach is followed in [GMT88] (see also [RMMM93]). Although not originally intended for the BSP model, the algorithm from [GMT88] (more precisely, its "m-contraction" phase) can be efficiently implemented on a BSPRAM. We sketch this implementation below.

The main idea of the method is to partition a tree of size n into edge-disjoint subtrees of size at most n/p, called *bridges*. Bridges possess an important characteristic property: each of them is attached to the tree by at most one leaf, and by the root (unless the bridge contains the root of the tree, and all its children). Figure 3.9 shows a tree partitioned into seven

bridges. Paper [GMT88] (see also [RMMM93]) shows that such partitioning always exists, and can be obtained by cutting the tree in at most $2p-1$ nodes. The partitioning can be computed by list contraction (specifically, all-prefix sums computation) on the Euler tour of the tree.

The BSPRAM algorithm is as follows.

Algorithm 7. *Tree contraction.*

Parameter: integer $n \geq p^2 \cdot \log p$ (respectively, $n \geq p^3 \cdot \log p$) for the randomised (respectively, deterministic) version of the algorithm.

Input: tree of size n.

Output: input tree contracted to a single node.

Description. The computation is performed on an EREW BSPRAM(p, g, l). Each processor reads an equal number of input nodes from the main memory. After this, the computation is performed in message-passing mode and proceeds in four stages.

First stage. The tree is partitioned into bridges. The partitioning is computed by several rounds of list contraction (specifically, all-prefix sums computation with varying basic operation) on the Euler tour of the tree. The list contraction is performed by Algorithms 5 or 6.

Second stage. A distribution of bridges across the processors is computed. By this distribution, each processor is assigned either a single bridge, or several bridges with a common root. The total size of the bridges assigned to any single processor is at most n/p. The distribution is computed by another all-prefix sums computation on the Euler tour of the tree.

Third stage. Each processor receives the assigned bridges and performs sequential tree contraction on each of them, reducing the bridges to their common root. This is made possible by the characteristic single-leaf attachment property of the bridges.

Fourth stage. The remaining tree of size p is collected in a single processor. This processor completes the contraction by local computation.

Cost analysis. The partitioning of the tree in the first stage and the distribution of the bridges in the second stage are computed by Algorithms 5 or 6. Their costs dominate (deterministically or with high probability) the cost of the remaining two stages. The costs of the whole algorithm (deterministic or expected with high probability) are

$$W_{\text{det/exp}} = O(n/p) \qquad H_{\text{det/exp}} = O(n/p) \qquad S_{\text{det/exp}} = O(\log p)$$

The algorithm is not communication-oblivious. Its slackness and granularity are the same as in Algorithms 5 or Algorithm 6: $\sigma_{\text{det/exp}} = n/p^2$, $\gamma = 1$. ■

Thus, tree contraction can be performed in the BSPRAM model with the aid of list contraction; little extra effort is required. The obtained algorithm for tree contraction has the same asymptotic costs as the list contraction algorithm employed.

Chapter 4

Dense matrix computation in the BSP model

4.1 Matrix-vector multiplication

Matrix-vector multiplication is a common operation in scientific computing, especially in iterative approximation methods. The general problem is defined as computation of the product $A \cdot b = c$, where A is an $n \times n$ matrix, and b, c are n-vectors over a semiring. The method consists in straightforward computation of the family of linear forms

$$c[i] = \sum_{j=1}^{n} A[i,j] \cdot b[j] \qquad 1 \leq i \leq n \tag{4.1}$$

Following (4.1), we need to set

$$c[i] \leftarrow 0 \quad \text{for } i = 1, \ldots, n \tag{4.2}$$

and then compute

$$c[i] \leftarrow c[i] + A[i,j] \cdot b[j] \quad \text{for all } i, j,\ 1 \leq i, j \leq n \tag{4.3}$$

Computation (4.3) for different pairs i, j is independent (although it requires concurrent reading from $b[j]$ and concurrent writing to $c[i]$), and therefore can be performed in parallel.

We assume that matrix A has been initially distributed across the processors, ignoring the BSP cost of such distribution. This assumption is natural in iterative approximation, where the cost of initial matrix redistribution can be amortised over a long series of iterations. By assuming that matrix A is predistributed, we concentrate on the cost of matrix-vector multiplication, rather than the cost of input/output.

The BSPRAM algorithm for matrix-vector multiplication is a straightforward adaptation of the BSP algorithm from [BM93, McC95]. Matrix A is

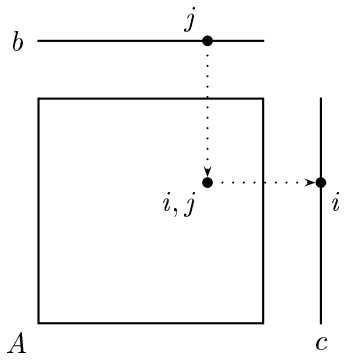


Figure 4.1: Matrix-vector multiplication dag

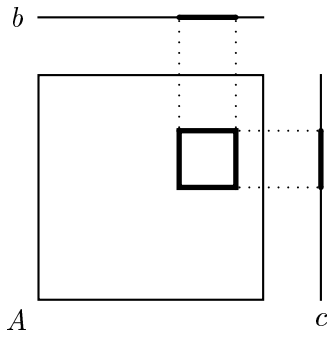


Figure 4.2: Matrix-vector multiplication in BSPRAM

represented by a square of size n in the integer plane (see Figure 4.1). Arrays b, c are represented by projections of the square A onto the coordinate axes. Computation of the product $A[i,j] \cdot b[j]$ requires the input from node $b[j]$, and the output to node $c[i]$. In order to provide a communication-efficient BSP algorithm, matrix A must be divided into p regular square blocks of size $n/p^{1/2}$ (see Figure 4.2),

$$A = \begin{pmatrix} A[[1,1]] & \cdots & A[[1,p^{1/2}]] \\ \vdots & \ddots & \vdots \\ A[[p^{1/2},1]] & \cdots & A[[p^{1/2},p^{1/2}]] \end{pmatrix} \tag{4.4}$$

Vectors b, c are divided into $p^{1/2}$ conforming regular intervals $b[[j]]$, $c[[i]]$ of size $n/p^{1/2}$. Computation (4.2), (4.3) can be expressed in terms of blocks as

$$c[[i]] \leftarrow 0 \quad \text{for } i = 1, \ldots, p^{1/2} \tag{4.5}$$

and then

$$c[[i]] \leftarrow c[[i]] + A[[i,j]] \cdot b[[j]] \quad \text{for all } i,j,\ 1 \le i,j \le p^{1/2} \tag{4.6}$$

The initial distribution of A is such that every processor holds a separate block $A[[i,j]]$, and computes the block product $A[[i,j]] \cdot b[[j]]$ sequentially by (4.2), (4.3). The algorithm is as follows.

Algorithm 8. *Matrix-vector multiplication.*

Parameters: integer $n \ge p^{1/2}$; $n \times n$ matrix A over a semiring, predistributed across the processors.

Input: n-vector b over a semiring.

Output: n-vector $c = A \cdot b$.

Description. The computation is performed on a CRCW BSPRAM(p,g,l). After the initialisation step (4.5), the computation proceeds in one superstep. Each processor performs the computation (4.6) for a particular pair i,j. In the input phase, the processor reads the block $b[[j]]$. Then it computes the product $A[[i,j]] \cdot b[[j]]$ by (4.2), (4.3). The computed block is then written to $c[[i]]$ in the main memory. Concurrent writing is resolved by addition of the written blocks to the previous content of $c[[i]]$. The resulting vector c is the product of A and b.

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

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

The algorithm is oblivious, with slackness and granularity $\sigma = \gamma = n^2/p^{1/2}$. ■

It can be shown that Algorithm 8 is an optimal algorithm for matrix-vector multiplication. The proof is omitted, due to its similarity to the optimality proof for matrix multiplication (Theorem 7 in Section 4.3).

4.2 Triangular system solution

Triangular systems of linear equations play an important role in scientific computation. One of their most common applications is in solution of general linear systems, where the system matrix is decomposed into a product of triangular factors, and then the resulting triangular systems are solved. The problem is formulated as follows: given a lower triangular matrix A and a vector c, find a vector b such that $A \cdot b = c$. For a nonsingular matrix A over a field, the solution is $b = A^{-1} \cdot c$. The problem can be formulated more generally over a semiring, using the closure operation. The generalised problem is: given a lower triangular matrix A and a vector c over a semiring, find

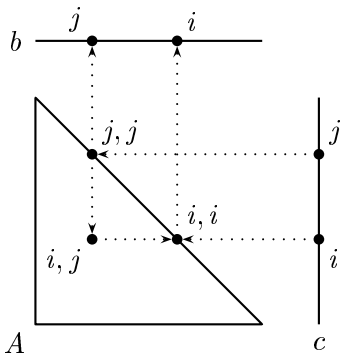
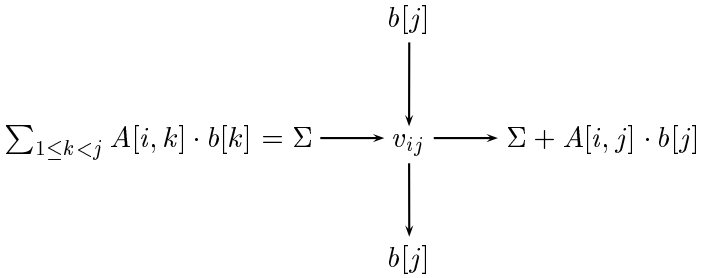


Figure 4.3: Triangular system solution dag

the vector $b = A^* \cdot c$, assuming that the matrix closure $A^* = I + A + A^2 + \cdots$ exists. The equivalence of the original and the generalised problem over a field follows from the identity $A^* = (I - A)^{-1}$.

Explicit computation of the closure A^* requires $\Theta(n^3)$ operations. However, there is no need to compute A^* explicitly: the standard substitution technique can be used to find $b = A^* \cdot c$ in sequential time $\Theta(n^2)$. Solution of a triangular system by substitution can be viewed as a dag computation. As before, matrix A is represented by the lower triangular part of a square of size n in the integer plane (see Figure 4.3). Arrays b, c are represented by projections of the square A onto the coordinate axes. Computation of the product $A[i,j] \cdot b[j]$, $i > j$, requires the input from node $A[j,j]$, and the output to node $A[i,i]$. A node $A[k,k]$ represents the computation of $b[k] = A[k,k]^* \cdot \big(c[k] + \sum_{1 \le j < k} A[k,j] \cdot b[j]\big)$. It requires the input from node $c[i]$, and from all products $A[k,j] \cdot b[j]$, $1 \le j < k$. The output of a node $A[k,k]$ is to node $b[i]$, and to all products $A[i,k] \cdot b[k]$, $k < i \le n$.

The above computation can also be arranged as a diamond dag $cube_2(n)$ (see e.g. [McC95]). Here, the action of a node v_{ij}, $i > j$, is



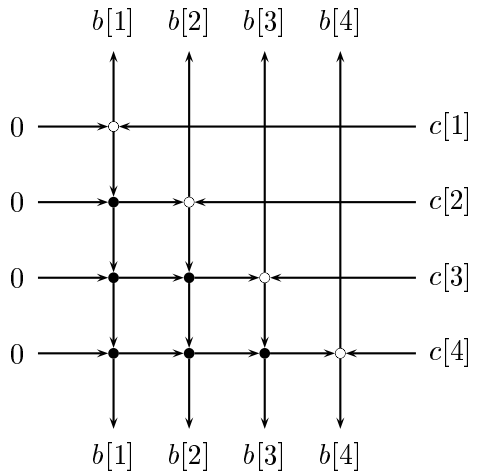
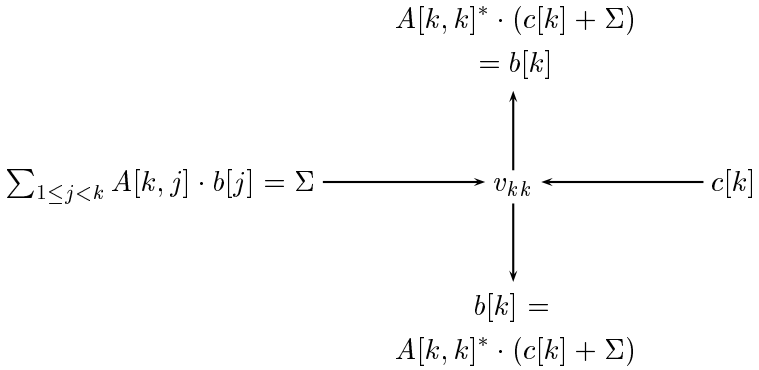


Figure 4.4: Triangular system solution by a diamond dag

and the action of a node v_{kk} is



Nodes v_{ij}, $i < j$, are not used. Figure 4.4 shows the resulting dag. It is similar to the diamond dag $cube_2(n)$, and can be computed by Algorithm 3 with BSP cost $W = O(n^2/p)$, $H = O(n)$, $S = O(p)$ (see Section 3.3).

An alternative approach to triangular system solution is recursion. The recursive algorithm works by dividing matrix A into square blocks of size $n/2$,

$$A = \begin{pmatrix} A_{11} & \\ A_{21} & A_{22} \end{pmatrix} \tag{4.7}$$

dividing vectors b, c into conforming intervals of size $n/2$,

$$b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \qquad c = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} \tag{4.8}$$

and then applying block substitution:

$$b_1 \leftarrow A_{11}^* \cdot c_1 \qquad b_2 \leftarrow A_{22}^* \cdot (c_2 + A_{21} \cdot b_1) \tag{4.9}$$

The procedure can be applied recursively to find the vectors $A_{11}^* \cdot c_1$ and $A_{22}^* \cdot (c_2 + A_{21} \cdot b_1)$. The resulting vector is

$$A^* \cdot c = \begin{pmatrix} A_{11}^* \cdot c_1 \\ A_{22}^* \cdot (c_2 + A_{21} \cdot A_{11}^* c_1) \end{pmatrix} \tag{4.10}$$

As in the previous section, we assume that matrix A has been initially distributed across the processors at no BSP cost. This assumption is natural in typical applications, such as solution of general linear systems, where the matrix A is obtained from a previous computation. By assuming that the matrix A is predistributed, we concentrate, as before, on the cost of solving the triangular system, rather than the cost of input/output.

We now describe the allocation of block triangular systems and block multiplication tasks in (4.9) to the BSPRAM processors. Initially, all p processors are available to compute the triangular system solution $A_{11}^* \cdot c_1$. There is no substantial parallelism between block triangular system solution and block multiplication tasks in (4.9); we can only exploit the parallelism within block multiplication. Therefore, the recursion tree is computed in depth-first order. In each level of recursion, every block multiplication in (4.9) is performed in parallel by all processors available at that level. Each triangular system in (4.9) is also solved in parallel by all processors available at that level, if the block size is large enough. When blocks become sufficiently small, triangular systems are solved sequentially by an arbitrarily chosen processor.

The initial distribution of A should allow one to perform the described computations without redistributing the matrix. The easiest way of achieving this is to partition matrix A into p^2 regular square blocks $A[[i,j]]$, $0 \le i,j < p$. A processor q can be assigned to hold e.g. all blocks $A[[i,q]]$ for $0 \le i < p$, or all blocks $A[[q,j]]$ for $0 \le j < p$, or all blocks $A[[i,j]]$ with $i - j = q$. The algorithm is as follows.

Algorithm 9. *Triangular system solution.*

Parameters: integer $n \ge p^{3/2}$; $n \times n$ matrix A over a semiring, predistributed across the processors.

Input: n-vector c over a semiring.

Output: n-vector $b = A^* \cdot c$.

Description. The computation is performed on a CRCW BSPRAM(p,g,l), and is defined by recursion on the size of the matrix and vectors. Denote the matrix size at the current level of recursion by m, keeping n for the original size. Let $n_0 = n/p$. Value n_0 is the threshold, at which the algorithm switches from parallel to sequential computation.

In each level of recursion, the matrix and vectors are divided into regular blocks of size $m/2$ as shown in (4.7), (4.8). Then, computation (4.9) is performed by the following schedule.

Small blocks. If $1 \leq m \leq n_0$, compute (4.9) on the processor that holds the current block.

Large blocks. If $n_0 < m \leq n$, compute b_1 by recursion. Then compute $A_{21} \cdot b_1$ by Algorithm 8. Compute $c_2 + A_{21} \cdot b_1$. Finally, compute $A_{22}^* \cdot (c_2 + A_{21} \cdot b_1)$ by recursion. Each of these computations is performed with all processors that are available without matrix redistribution.

Cost analysis. The values for $W = W_p(n)$, $H = H_p(n)$, $S = S_p(n)$ can be found from the following recurrence relations:

	$n_0 < m \leq n$	$m = n_0$
$W_q(m) =$	$2 \cdot W_{q/2}(m/2) + O(m^2/q)$	$O(n_0^2)$
$H_q(m) =$	$2 \cdot H_{q/2}(m/2) + O(m/q^{1/2})$	$O(n_0)$
$S_q(m) =$	$2 \cdot S_q(m/2) + O(1)$	$O(1)$

as

$$W = O(n^2/p) \qquad H = O(n) \qquad S = O(p)$$

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

The above analysis shows that the recursive algorithm for triangular system solution has the same BSP cost as the diamond dag algorithm. The use of block multiplication in the recursive algorithm does not lead to an improvement in communication cost.

We now show that the asymptotic BSP cost of Algorithm 9 cannot be reduced by any computation of the substitution dag (Figure 4.3).

Theorem 6. *Any BSPRAM(p, g, l) computation of the triangular system substitution dag with $W = O(n^2/p)$ requires (i) $W = \Theta(n^2/p)$, (ii) $H = \Omega(n)$, (iii) $S = \Omega(p)$.*

Proof. (i) Trivial.

(ii) Let $m_0 = 0$. Let q_0 be the first processor computing the node v_{00}. Let m_1 be the first row of G not containing a node computed by processor q_0 (if every row contains a node computed by q_0, then $m_1 = n$). Define G_0 as a subdag of G consisting of rows i with $0 \leq i \leq m_1$, and F_1 as a subdag of G consisting of columns j with $m_1 \leq j \leq n-1$. Let q_1 be the first processor computing the node $v_{m_1 m_1}$. Let m_2 be the first row of F_1 not containing a node computed by processor q_1. Define G_1 as a subdag of F_1 consisting of rows i with $m_1 \leq i \leq m_2$, and F_2 as a subdag of F_1 consisting of columns j with $m_2 \leq j \leq n-1$. Repeat this process until some current $m_{r+1} = n$ (and therefore F_{r+1} is empty). We have obtained a sequence of subdags $G_0, \ldots, G_r$ along the main diagonal of the dag G (see Figure 4.5). The computation of each subdag can start only after each node in the previous

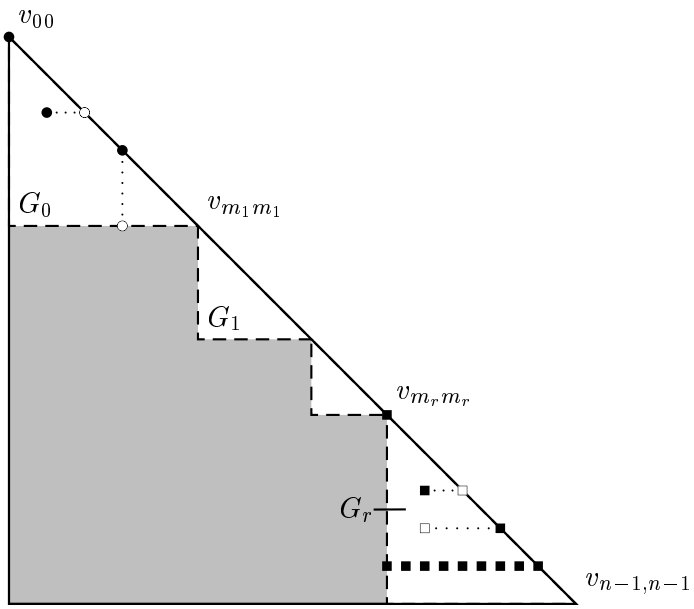


Figure 4.5: Proof of Theorem 6, part (*ii*)

subdag has been computed at least once. For the purpose of a lower bound, we will ignore all computation in a subdag after the highest-indexed node has been computed once. We may also assume that each computed value is used at least once.

Consider a subdag G_s, $0 \leq s < r$. For each k, $m_s \leq k < m_{s+1}$, there are two cases:

- v_{kk} is not computed by processor q_s. By construction of G_s, there is some j, $m_s \leq j < k$, such that v_{kj} is computed by processor q_s. Therefore, processor q_s must communicate the value computed in v_{kj} to one of the processors computing v_{kk}.

- v_{kk} is computed by processor q_s. By construction of G_s, the node $v_{m_{s+1}k}$ is not computed by processor q_s. Therefore, processor q_s must communicate the value computed in v_{kk} to the processor computing $v_{m_{s+1}k}$.

Thus, for each k, processor q_s must send a distinct value, therefore the communication cost of computing G_s is at least $m_{s+1} - m_s$.

Now consider the subdag G_r. For each k, $m_r \leq k < n$, there are three cases:

- v_{kk} is not computed by processor q_r. Similarly to the first case above, processor q_r must communicate the value computed in some v_{kj}, $m_r \leq j < k$, to one of the processors computing v_{kk}.

- v_{kk} is computed by processor q_r, but for some j, $m_r \leq j < k$, the node v_{kj} is not computed by processor q_r. In this case, the value computed in v_{kj} must be communicated to processor q_r in order to compute v_{kk}.
- v_{kj} is computed by processor q_r for all j, $m_r \leq j < k$. Since $W = O(n^2/p)$, the total number of such values k cannot exceed $n/p^{1/2}$.

Thus, for each k, except at most $n/p^{1/2}$ values, processor q_s must send or receive a distinct value, therefore the communication cost of computing G_r is at least $n - m_r - n/p^{1/2}$.

Since for every s, the computation of G_s must be completed before the computation of G_{s+1} can start, the total communication cost is at least

$$(m_1 - m_0) + (m_2 - m_1) + \cdots + (n - m_r) - n/p^{1/2} = n - n/p^{1/2} = \Omega(n)$$

(*iii*) The proof is a two-dimensional version of the proof for Theorem 5, part (*iii*). Nodes of the main diagonal v_{ii}, $0 \leq i < n$, first computed in each particular superstep, form a consecutive segment. We denote the sizes of these segments by m_s, $0 \leq s < S$, where $\sum_{0 \leq s < S} m_s = n$. Since the communication within a superstep is not allowed, all nodes in the diagonal square block of size m_s spanned by the segment s must be computed by the processor that first computes the highest-indexed node of the block. The local computation cost of a block is therefore m_s^2. The total cost of local computation is bounded from below by the sum over all supersteps: $\sum_{0 \leq s < S} m_s^2$. By Cauchy's inequality,

$$n = \sum_{0 \leq s < S} m_s = \sum_{0 \leq s < S} 1 \cdot m_s \leq S^{1/2} \cdot \left(\sum_{0 \leq s < S} m_s^2 \right)^{1/2} \leq S^{1/2} \cdot W^{1/2}$$

Hence by assumption $S \geq n^2/W = \Omega(p)$. ■

Note that the recursive block substitution used in Algorithm 9 defines a dag different from the ordinary substitution dag in Figure 4.3. Hence, Theorem 6 does not cover all standard methods of triangular system solution in BSP, in particular the method used by Algorithm 9 itself. However, it may be possible to extend the theorem to a more general class of dags, including the ordinary substitution and the recursive block substitution dag, as well as the diamond dag. The standard algorithms for all the above dags have similar BSP costs, which suggests that these algorithms may be optimal for triangular system solution under the new extended definition.

4.3 Matrix multiplication

In this section we describe a BSPRAM algorithm for one of the most common problems in scientific computation: dense matrix multiplication. We deal

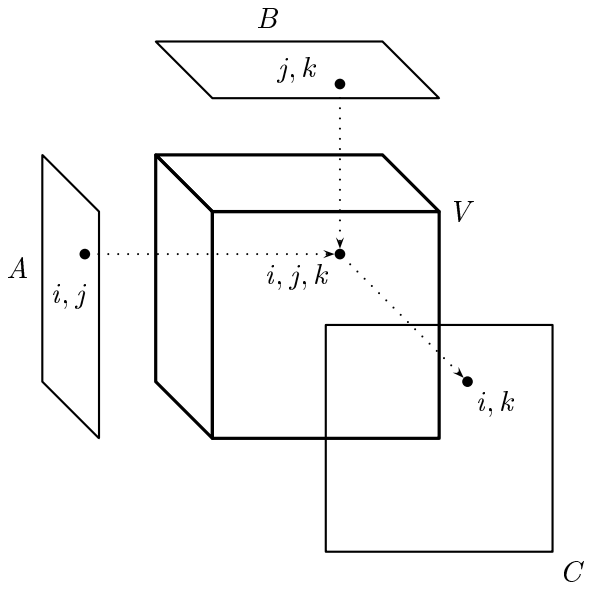


Figure 4.6: Matrix multiplication dag

with the general problem of computing the matrix product $A \cdot B = C$, where A, B, C are $n \times n$ matrices over a semiring.

We aim to parallelise the standard $\Theta(n^3)$ method, asymptotically optimal for sequential matrix multiplication over a general semiring (see [HK71]). The method consists in straightforward computation of the family of bilinear forms

$$C[i,k] = \sum_{j=1}^{n} A[i,j] \cdot B[j,k] \qquad 1 \leq i,k \leq n \tag{4.11}$$

Following (4.11), we need to set

$$C[i,k] \leftarrow 0 \quad \text{for } i,k = 1,\ldots,n \tag{4.12}$$

and then compute

$$V[i,j,k] \leftarrow A[i,j] \cdot B[j,k] \qquad C[i,k] \leftarrow C[i,k] + V[i,j,k] \tag{4.13}$$

for all i,j,k, $1 \leq i,j,k \leq n$. Computation (4.13) for different triples i,j,k is independent (although it requires concurrent reading from $A[i,j]$ and $B[j,k]$, and concurrent writing to $C[i,k]$), and therefore can be performed in parallel.

The BSPRAM algorithm implementing this method is derived from the BSP algorithm due to McColl and Valiant, described in [McC95, McC96c]. The algorithm combines the idea of two-phase broadcast (see Section 2.2) with symmetric three-dimensional problem partitioning, previously used e.g.

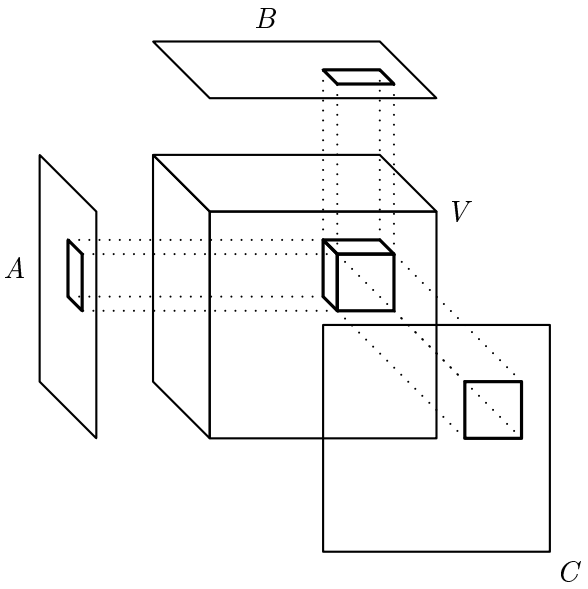


Figure 4.7: Matrix multiplication in BSPRAM

in [ACS90] (see also [ABG+95] for further references and experimental results). The algorithm works by a straightforward partitioning of the problem dag. Array V is represented as a cube of volume n^3 in integer three-dimensional space (see Figure 4.6). Arrays A, B, C are represented as projections of the cube V onto the coordinate planes. Computation of the node $V[i,j,k]$ requires the input from nodes $A[i,j]$, $B[j,k]$, and the output to node $C[i,k]$. 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 Figure 4.7). Such partitioning induces a partition of the matrices A, B, C into $p^{2/3}$ regular square blocks of size $n/p^{1/3}$,

$$A = \begin{pmatrix} A[[1,1]] & \cdots & A[[1,p^{1/3}]] \\ \vdots & \ddots & \vdots \\ A[[p^{1/3},1]] & \cdots & A[[p^{1/3},p^{1/3}]] \end{pmatrix} \tag{4.14}$$

and similarly for B, C (see Figure 4.7). Computation (4.12), (4.13) can be expressed in terms of blocks as

$$C[[i,k]] \leftarrow 0 \quad \text{for } i,k = 1,\ldots,p^{1/3} \tag{4.15}$$

and then

$$V[[i,j,k]] \leftarrow A[[i,j]] \cdot B[[j,k]] \qquad C[[i,k]] \leftarrow C[[i,k]] + V[[i,j,k]] \tag{4.16}$$

for all i,j,k, $1 \le i,j,k \le p^{1/3}$. Each processor computes a block product $V[[i,j,k]] = A[[i,j]] \cdot B[[j,k]]$ sequentially by (4.12), (4.13). The algorithm is as follows.

Algorithm 10. *Matrix multiplication.*

Parameter: integer $n \geq p^{1/3}$.

Input: $n \times n$ matrices A, B over a semiring.

Output: $n \times n$ matrix $C = A \cdot B$.

Description. The computation is performed on a CRCW BSPRAM(p, g, l). After the initialisation step (4.15), the computation proceeds in one superstep. Each processor performs the computation (4.16) for a particular triple i, j, k. In the input phase, the processor reads the blocks $A[[i, j]]$ and $B[[j, k]]$. Then it computes the product $V[[i, j, k]] = A[[i, j]] \cdot B[[j, k]]$ by (4.12), (4.13). The block $V[[i, j, k]]$ is then written to $C[[i, k]]$ in the main memory. Concurrent writing is resolved by addition of the written blocks. The resulting matrix C is the product of A and B.

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

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

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

We now prove that Algorithm 10 is an optimal parallel realisation of the standard matrix multiplication algorithm. The following theorem was suggested by [Pat93].

Theorem 7. *Any BSPRAM(p, g, l) computation of the standard matrix multiplication dag requires* (i) $W = \Omega(n^3/p)$, (ii) $H = \Omega(n^2/p^{2/3})$, (iii) $S = \Omega(1)$.

Proof. (i), (iii) Trivial.

(ii) Since n^3 nodes are computed by p processors, there is a processor that computes at least n^3/p nodes. We apply the discrete Loomis–Whitney inequality (see Appendix A) to this set of nodes. Since there are at least n^3/p nodes in the set, one of the three projections of the set must contain at least $n^2/p^{2/3}$ nodes, therefore $H = \Omega(n^2/p^{2/3})$. ■

Algorithm 10 will serve us as a building block for more advanced matrix algorithms developed in the following sections.

4.4 Fast matrix multiplication

In Section 4.3 we considered the problem of matrix multiplication over a semiring. As mentioned before, the standard $\Theta(n^3)$ sequential algorithm is optimal for a general semiring. However, this is not so for commutative

rings with unit, which allow "fast" matrix multiplication algorithms. The first such algorithm was proposed by Strassen in his groundbreaking paper [Str69]. Since then, much work has been done on the complexity of matrix multiplication over a commutative ring with unit. However, no lower bound asymptotically better than the trivial $\Omega(n^2)$ has been found, nor is there any indication that the current $O(n^{2.376})$ algorithm from [CW90] is close to optimal.

The natural computational model for matrix multiplication over a commutative ring with unit is the model of arithmetic circuits. It is not difficult to see (see e.g. [HK71]) that without loss of generality, the model for matrix multiplication can be restricted to a special class of circuits, called bilinear. Let A, B, C be $N \times N$ matrices over a commutative ring with unit. A *bilinear circuit* for the matrix product $A \cdot B = C$ computes a family of bilinear forms

$$C[i,k] = \sum_{r=1}^{R} \gamma_{ik}^{(r)} \left(\sum_{i,j=1}^{N} \alpha_{ij}^{(r)} A[i,j] \right) \left(\sum_{j,k=1}^{N} \beta_{jk}^{(r)} B[j,k] \right) \tag{4.17}$$

for $1 \le i,k \le N$, where $\alpha_{ij}^{(r)}$, $\beta_{jk}^{(r)}$, $\gamma_{ik}^{(r)}$ are constant elements of the ring. We assume that all R terms in (4.17) are nontrivial, i.e. for each r, there are some $\alpha_{ij}^{(r)} \neq 0$, $\beta_{jk}^{(r)} \neq 0$ and $\gamma_{ik}^{(r)} \neq 0$. The number R is called the *multiplicative complexity* of the bilinear circuit.

We represent the bilinear circuit (4.17) by a dag that we call a *bilinear dag*. Each of the terms in (4.17) is represented by a node $v^{(r)}$, $1 \le r \le R$. Computation of the node $v^{(r)}$ requires the input of $A[i,j]$ for all i,j such that $\alpha_{ij}^{(r)} \neq 0$, and of $B[j,k]$ for all j,k such that $\beta_{jk}^{(r)} \neq 0$, as well as the output of $C[i,k]$ for all i,k such that $\gamma_{ik}^{(r)} \neq 0$.

Following (4.17), we need to set

$$C[i,k] \leftarrow 0 \quad \text{for } i,k = 1,\ldots,n \tag{4.18}$$

and then compute

$$\begin{aligned}
&x^{(r)} \leftarrow 0; y^{(r)} \leftarrow 0 \\
&x^{(r)} \leftarrow x^{(r)} + \alpha_{ij}^{(r)} A[i,j] && \text{for } i,j = 1,\ldots,N \\
&y^{(r)} \leftarrow y^{(r)} + \beta_{jk}^{(r)} B[j,k] && \text{for } j,k = 1,\ldots,N \\
&v^{(r)} \leftarrow x^{(r)} \cdot y^{(r)} \\
&C[i,k] \leftarrow C[i,k] + \gamma_{ik}^{(r)} v^{(r)} && \text{for } i,k = 1,\ldots,N
\end{aligned} \tag{4.19}$$

for all r, $1 \le r \le R$. Computation (4.19) for different values of r is independent (although it requires concurrent reading from $A[i,j]$, $B[j,k]$, and concurrent writing to z_{ik}), therefore it can be performed in parallel.

A bilinear circuit based on the standard definition of matrix product (4.11) has multiplicative complexity $R = N^3$. The first non-standard bilinear circuit for matrix multiplication with $N = 2$ and $R = 7$ was proposed in [Str69]. Paper [HK71] shows that for any arithmetic circuit solving the matrix multiplication problem over a commutative ring with unit in r non-scalar multiplications or divisions, there exists a bilinear circuit solving this problem with multiplicative complexity at most $2r$.

Any bilinear circuit for multiplying matrices of size $N \times N$ can be applied to matrices of size $n \geq N$. The matrices A, B, C are divided into regular square blocks of size n/N,

$$A = \begin{pmatrix} A[[1,1]] & \cdots & A[[1,N]] \\ \vdots & \ddots & \vdots \\ A[[N,1]] & \cdots & A[[N,N]] \end{pmatrix} \tag{4.20}$$

and similarly for B, C. The computation (4.18), (4.19) can be expressed in terms of blocks as

$$C[[i,k]] \leftarrow 0 \quad \text{for } i,k = 1,\ldots,N \tag{4.21}$$

and then

$$\begin{aligned} &X^{(r)} \leftarrow 0; Y^{(r)} \leftarrow 0 \\ &X^{(r)} \leftarrow X^{(r)} + a_{ij}^{(r)} A[[i,j]] && \text{for } i,j = 1,\ldots,N \\ &Y^{(r)} \leftarrow Y^{(r)} + b_{jk}^{(r)} B[[j,k]] && \text{for } j,k = 1,\ldots,N \\ &V^{(r)} \leftarrow X^{(r)} \cdot Y^{(r)} \\ &C[[i,k]] \leftarrow C[[i,k]] + c_{ik}^{(r)} V^{(r)} && \text{for } i,k = 1,\ldots,N \end{aligned} \tag{4.22}$$

for all r, $1 \leq r \leq R$. The procedure is applied recursively to compute the block product $V^{(r)} = X^{(r)} \cdot Y^{(r)}$. The resulting algorithm has sequential complexity $\Theta(n^\omega)$, where $\omega = \log_N R$.

A BSP version of the algorithm was proposed in [McC96b] (see also [KHSJ95, GvdG96, GHSJ96]). The recursion tree is computed in breadth-first order. The algorithm uses a data distribution that allows one to compute the linear forms in (4.22) in a constant number of supersteps. Each of the matrices A, B, C is divided into regular square submatrices of size $n_0 \times n_0$, where $n_0 = n/p^{1/\omega}$. The matrices are distributed across the processors, so that the distributions of each of the above submatrices are even and identical. Examples of a suitable distribution of A, B, C are the cyclic distribution, or any block-cyclic distribution with square blocks of size at most $n_0/p^{1/2}$.

The described data distribution allows one to compute the linear forms in (4.22) without communication, until the current matrix size is reduced to

n_0, and p independent matrix multiplication subproblems are generated. At this point the data are redistributed, so that each subproblem can be solved sequentially. The algorithm is as follows:

Algorithm 11. *Fast matrix multiplication.*

Parameter: integer $n \geq p^{1/\omega}$.

Input: $n \times n$ matrices A, B over a commutative ring with unit.

Output: $n \times n$ matrix $C = A \cdot B$.

Description. The computation is performed on a CRCW BSPRAM(p, g, l), and is defined by recursion on the size of the matrix. We denote the matrix size at the current level of recursion by m, keeping n for the original size. Let $n_0 = n/p^{1/\omega}$. Value n_0 is the threshold, at which the data are redistributed among the processors.

In each level of recursion, the matrix is divided into N^2 regular square blocks of size m/N as shown in (4.20). We perform the initialisation (4.21), and then the computation (4.22) by the following schedule.

Small blocks. If $1 \leq m \leq n_0$, compute (4.22) sequentially on the processor where the data are held.

Large blocks. If $n_0 < m \leq n$, generate R multiplication subproblems by executing the first three lines of (4.22) in parallel for all r. Solve the multiplication subproblems in parallel by R simultaneous recursive calls. Compute the result by executing the final line of (4.22) in parallel for all r. The data distribution ensures that the linear steps can be performed without communication.

In the above description, the data are implicitly redistributed when the matrix size m passes the threshold n_0. At this stage, the recursion tree is evaluated in breadth-first order. Therefore, the redistribution occurs only twice — first on the down-sweep, then on the up-sweep of the recursion tree.

Cost analysis. The values for $W = W(n)$, $H = H(n)$, $S = S(n)$ can be found from the following recurrence relations:

	$n_0 < m \leq n$	$m = n_0$
$W(m) =$	$R \cdot W(m/N)$	$O(n_0^\omega/p)$
$H(m) =$	$R \cdot H(m/N)$	$O(n_0^2/p)$
$S(m) =$	$S(m/N)$	$O(1)$

as

$$W = O(n^\omega/p) \qquad H = O\big(n^2/p^{2\omega^{-1}}\big) \qquad S = O(1)$$

The algorithm is oblivious, with slackness $\sigma = n^2/p^{2\omega^{-1}}$ and granularity $\gamma = n^2/p^{2\omega^{-1}+1}$. ■

We now prove that Algorithm 11 is an optimal parallel realisation of the fast matrix multiplication algorithm. In contrast with standard matrix multiplication, we do not have any single dag underlying the computation. A statement that any BSP implementation of a bilinear circuit with parameter ω would require $H = \Omega(N^2/p^{2\omega^{-1}})$ is obviously invalid, since we might be able to emulate such a circuit by a circuit with $\omega_1 < \omega$, by introducing "spurious" terms in (4.17) (e.g., duplicating other terms). The resulting circuit could be computed in $H = O(N^2/p^{2\omega_1^{-1}}) = o(N^2/p^{2\omega^{-1}})$. Thus, a lower bound on communication cost of a general bilinear matrix multiplication circuit is closely related to the lower bound on its computation cost, which is beyond the reach of current complexity theory. Therefore, we restrict our analysis to circuits obtained by recursive application of a fixed basic circuit (4.17). The analysis can be easily extended to the case where different levels of recursion are defined by different basic circuits of the form (4.17), but all such circuits must have a fixed maximum size N and a fixed minimum multiplicative complexity R (and therefore a fixed minimum exponent ω).

The dag for recursive multiplication of matrices of size n based on the circuit (4.17) consists of $\log n / \log N$ levels. Each level is formed from disjoint copies of the bilinear dag corresponding to the basic circuit. Algorithm 11 performs input/output at level 0, and data redistribution at level $\log p / \log R$.

We now prove the optimality of Algorithm 11.

Theorem 8. *Any BSPRAM(p, g, l) computation of the recursive matrix multiplication dag based on* (4.17) *requires* (i) $W = \Omega(n^\omega/p)$, (ii) $H = \Omega(n^2/p^{2\omega^{-1}})$, (iii) $S = \Omega(1)$.

Proof. (i), (iii) Trivial.

(ii) Induction on p and n.

Induction base ($p = 1$, arbitrary n). Trivial.

Inductive step ($p \to R \cdot p$, $n \to N \cdot n$). The outermost level of the dag consists of n^2 disjoint copies of the basic dag. The rest of the dag consists of R disjoint copies of the fast matrix multiplication dag of size n. For each of these copies, the communication cost of Algorithm 11 is optimal. The outermost level is computed by Algorithm 11 without communication, therefore the overall communication is optimal. ■

It should be noted that for standard matrix multiplication (a basic dag with $N = 2$, $R = 8$), Theorem 8 is not a replacement for the optimality proof of Algorithm 10 (Theorem 7). This is because the computation of the standard matrix multiplication dag (Figure 4.6) does not have to follow the recursive pattern of fast matrix multiplication. Therefore, the statement of Theorem 7 is more general than Theorem 8 applied to the dag based on standard 2×2 matrix multiplication.

4.5 Gaussian elimination without pivoting

This section and the next describe a BSPRAM approach to Gaussian elimination, a method primarily used for direct solution of linear systems of equations. More generally, Gaussian elimination and its variations are applied to a broad spectrum of numerical, symbolic and combinatorial problems.

In this section we consider the simplest form of Gaussian elimination, which does not involve the search for pivots. This basic form of elimination is not guaranteed to produce correct result, or terminate at all, when performed on arbitrary matrices. However, it works well for matrices over some particular domains, such as closed semirings, or for matrices of some particular types, such as symmetric positive definite matrices over real numbers. We will consider pivoting in Sections 4.6 and 4.7.

Gaussian elimination can be described in many ways. In this section we consider it as LU decomposition of a real matrix. Chapter 5 presents another form of Gaussian elimination without pivoting, used for algebraic path computation and matrix inversion.

Let A be an $n \times n$ real diagonally dominant or symmetric positive definite matrix. The LU decomposition of A is $A = L \cdot U$, where L is an $n \times n$ unit lower triangular, and R is an $n \times n$ upper triangular matrix. This decomposition can be computed in sequential time $O(n^3)$ by plain Gaussian elimination, or in time $O(n^\omega)$ by block Gaussian elimination, using fast matrix multiplication.

The parallel complexity of Gaussian elimination has been extensively studied in many models of parallel computation. Paper [McC95] proposes to reduce the problem to the computation of a cube dag $cube_3(n)$. The reduction is similar to the one described in Section 4.2 for triangular system solution. The BSP cost of the resulting computation is $W = O(n^3/p)$, $H = O(n^2/p^{1/2})$, $S = O(p^{1/2})$ (see Section 3.3). The cube dag method is straightforward for LU decomposition, and can be easily adapted to other forms of Gaussian elimination, such as QR decomposition by Givens rotations.

A lower communication cost for LU decomposition can be achieved by an alternative algorithm, based on recursive block Gauss–Jordan elimination (see e.g. [GPS90, DHS95]). This standard method was suggested as a means of reducing the communication cost in [ACS90] (for the transitive closure problem). Given a nonsingular matrix A, the algorithm produces the LU decomposition $A = L \cdot U$, together with the inverse matrices L^{-1} and U^{-1}. The algorithm works by dividing the matrices A, L, U into square blocks of size $n/2$,

$$A = L \cdot U: \qquad \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \cdot \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ \cdot & U_{22} \end{pmatrix} \tag{4.23}$$

where the dot $\cdot$ indicates a zero block. First we find the LU decomposition

of the block $A_{11} = L_{11} \cdot U_{11}$, along with the inverse blocks L_{11}^{-1} and U_{11}^{-1}, by applying the algorithm recursively. Then we apply block Gauss–Jordan elimination to find the blocks L_{21}, U_{12}:

$$L_{21} \leftarrow A_{21} \cdot U_{11}^{-1} \qquad U_{12} \leftarrow L_{11}^{-1} \cdot A_{12} \tag{4.24}$$

A second recursive application of the algorithm yields the LU decomposition $A_{22} - L_{21} \cdot U_{12} = L_{22} \cdot U_{22}$, and the inverse blocks L_{22}^{-1}, U_{22}^{-1}. We complete the computation by taking

$$L^{-1} = \begin{pmatrix} L_{11}^{-1} & \cdot \\ -L_{22}^{-1} \cdot L_{21} \cdot L_{11}^{-1} & L_{22}^{-1} \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} U_{11}^{-1} & -U_{11}^{-1} \cdot U_{12} \cdot U_{22}^{-1} \\ \cdot & U_{22}^{-1} \end{pmatrix} \tag{4.25}$$

We now describe the allocation of block LU decomposition tasks and block multiplication tasks in (4.24)–(4.25) to the BSPRAM processors. Initially, all p processors are available to compute the LU decomposition. There is no substantial parallelism between block decomposition and block multiplication tasks in (4.24)–(4.25); we can only exploit the parallelism within block multiplication. Therefore, the recursion tree has to be computed in depth-first order. In each level of recursion, every block multiplication in (4.24)–(4.25) is performed in parallel by all processors available at that level. Each block LU decomposition is also performed in parallel by all processors available at that level, if the block size is large enough. When blocks become sufficiently small, block LU decomposition is computed sequentially by an arbitrarily chosen processor.

The depth at which the algorithm switches from p-processor to single-processor computation can be varied. This variation allows us to trade off the costs of communication and synchronisation in a certain range. In order to account for this tradeoff, we introduce a real parameter α, controlling the depth of parallel recursion. The algorithm is as follows.

Algorithm 12. *Gaussian elimination without pivoting.*

Parameters: integer $n \geq p$; real number α, $\alpha_{\min} = 1/2 \leq \alpha \leq 2/3 = \alpha_{\max}$.

Input: $n \times n$ real matrix A; we assume that A is diagonally dominant or symmetric positive definite.

Output: decomposition $A = L \cdot U$, where L is an $n \times n$ unit lower triangular matrix, and U is an $n \times n$ upper triangular matrix.

Description. The computation is performed on a CRCW BSPRAM(p, g, l), and is defined by recursion on the size of the matrix. Denote the matrix size at the current level of recursion by m, keeping n for the original size. Let $n_0 = n/p^{\alpha}$. Value n_0 is the threshold, at which the algorithm switches from parallel to sequential computation.

In each level of recursion, the matrix is divided into regular square blocks of size $m/2$ as shown in (4.23). Then, computation (4.24)–(4.25) is performed by the following schedule.

Small blocks. If $1 \leq m \leq n_0$, choose an arbitrary processor from all currently available, and compute (4.24)–(4.25) on that processor.

Large blocks. If $n_0 < m \leq n$, compute L_{11}, U_{11}, L_{11}^{-1}, U_{11}^{-1} by recursion. Then compute L_{21}, U_{12} and $A_{22} - L_{21} \cdot U_{12}$ by Algorithm 10. After that, compute L_{22}, U_{22}, L_{22}^{-1}, U_{22}^{-1} by recursion. Finally, compute $-L_{22}^{-1} \cdot L_{21} \cdot L_{11}^{-1}$ and $-U_{11}^{-1} \cdot U_{12} \cdot U_{22}^{-1}$ by Algorithm 10. Each of these computations is performed with all available processors. The result is the decomposition (4.23).

Cost analysis. The values for $W = W(n)$, $H = H(n)$, $S = S(n)$ can be found from the following recurrence relations:

	$n_0 < m \leq n$	$m = n_0$
$W(m) =$	$2 \cdot W(m/2) + O(m^3/p)$	$O(n_0^3)$
$H(m) =$	$2 \cdot H(m/2) + O(m^2/p^{2/3})$	$O(n_0^2)$
$S(m) =$	$2 \cdot S(m/2) + O(1)$	$O(1)$

as

$$W = O(n^3/p) \qquad H = O(n^2/p^\alpha) \qquad S = O(p^\alpha)$$

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

For $\alpha = \alpha_{\min} = 1/2$, the cost of Algorithm 12 is $W = O(n^3/p)$, $H = O(n^2/p^{1/2})$, $S = O(p^{1/2})$. This is asymptotically equal to the BSP cost of the cube dag method from [McC95]. For $\alpha = \alpha_{\max} = 2/3$, the cost of Algorithm 12 is $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, $S = O(p^{2/3})$. In this case, the communication cost is as low as in matrix multiplication (Algorithm 10). This improvement in communication efficiency is offset by a reduction in synchronisation efficiency. For large n, the communication cost of Algorithm 12 dominates the synchronisation cost, and therefore the communication improvement should outweigh the loss of synchronisation efficiency. This justifies the use of Algorithm 12 with $\alpha = \alpha_{\max} = 2/3$. Smaller values of α, or the cube dag algorithm, should be considered when the problem is moderately sized.

As in triangular system solution (Section 4.2), Gaussian elimination without pivoting cannot be defined as a computation of any particular dag. The cube dag method, ordinary elimination and block recursive elimination produce different dags. A lower bound on synchronisation cost can be proved for a general class of dags, including the three subclasses above. A proof, identical to the proof for the cube dag (Theorem 5, part (*iii*)), gives

the bound $S = \Omega(p^{1/2})$, provided that $W = O(n^3/p)$. A lower bound on the communication cost can be obtained by the standard method of reducing the matrix multiplication problem to LU decomposition. For any $n \times n$ real matrices A, B, the product $A \cdot B$ can be computed by LU decomposition of a $2n \times 2n$ matrix:

$$\begin{pmatrix} I & B \\ A & I \end{pmatrix} = \begin{pmatrix} I & \cdot \\ A & I \end{pmatrix} \begin{pmatrix} I & B \\ \cdot & I - A \cdot B \end{pmatrix}$$

Therefore, the lower bound $H = \Omega(n^2/p^{2/3})$ for standard matrix multiplication (Theorem 7) holds also for standard Gaussian elimination without pivoting, which must be appropriately defined to exclude Strassen-type methods.

Fast matrix multiplication can be used instead of standard matrix multiplication for computing block products. The modified algorithm is as follows.

Algorithm 13. *Fast Gaussian elimination without pivoting.*

Parameters: integer $n \geq p^{3/\omega}$; real number α, $\alpha_{\min} = 1/(\omega - 1) \leq \alpha \leq 2/\omega = \alpha_{\max}$.

Input: $n \times n$ real matrix A; we assume that A is diagonally dominant or symmetric positive definite.

Output: decomposition $A = L \cdot U$, where L is an $n \times n$ unit lower triangular matrix, and U is an $n \times n$ upper triangular matrix.

Description. The computation is identical to Algorithm 12, except that block multiplication is performed by Algorithm 11, rather than Algorithm 10. As before, $n_0 = n/p^\alpha$ is the threshold between parallel and sequential computation.

Cost analysis. The values for $W = W(n)$, $H = H(n)$, $S = S(n)$ can be found from the following recurrence relations:

	$n_0 < m \leq n$	$m = n_0$
$W(m) =$	$2 \cdot W(m/2) + O(m^\omega/p)$	$O(n_0^\omega)$
$H(m) =$	$2 \cdot H(m/2) + O\left(m^2/p^{2\omega^{-1}}\right)$	$O(n_0^2)$
$S(m) =$	$2 \cdot S(m/2) + O(1)$	$O(1)$

as

$$W = O(n^\omega/p) \qquad H = O(n^2/p^\alpha) \qquad S = O(p^\alpha)$$

The algorithm is oblivious, with slackness and granularity $\sigma = \gamma = n^2/p^{2\omega^{-1}}$. ■

As ω approaches the value of 2, the range of parameter α becomes tighter. If an $O(n^2)$ matrix multiplication algorithm is eventually discovered, the tradeoff between H and S will disappear.

4.6 Nested block pivoting and Givens rotations

In this section we extend the results of the previous section, obtaining an efficient BSP algorithm for certain forms of Gaussian elimination with pivoting.

Let A be an $n \times n$ matrix over a finite field. We assume that the inverse of a nonzero field element can be computed in time $O(1)$. Plain Gaussian elimination without pivoting may fail to find the LU decomposition of A, because some diagonal elements may be zero initially, or become zero during elimination. Block Gaussian elimination without pivoting may fail for a similar reason, if some diagonal blocks are singular or become singular during elimination. A particular feature of computation over a finite field is that any nonzero element, or any nonsingular block, can serve as a pivot. As we show below, this allows us to use a restricted version of pivoting, proposed in [Sch73] (see also [BCS97, section 16.5]). We call this technique *nested block pivoting*, since its approach is to find a sequence of nested nonsingular blocks, or, if the original matrix is singular, a sequence of nested blocks of maximum possible rank. A similar approach applies to computation of the QR decomposition of a real matrix by Givens rotations.

Another pivoting method suitable for block triangular decomposition has been proposed in [BH74]. Since the approach of [BH74] requires a search for the pivot along a matrix row, its BSP cost is higher than the cost of nested block pivoting.

To describe nested block pivoting, we consider Gaussian elimination on rectangular matrices of a special form. Let $\binom{A}{V}$ be a $2n \times n$ matrix over a finite field. Here A is an arbitrary $n \times n$ matrix, and V an upper triangular $n \times n$ matrix. Matrices A, V may not have full rank. The problem consists in finding a full-rank $2n \times 2n$ transformation matrix $\left(\begin{smallmatrix} D & E \\ F & G \end{smallmatrix}\right)$, and an $n \times n$ upper triangular matrix U, such that

$$\begin{pmatrix} D & E \\ F & G \end{pmatrix} \begin{pmatrix} A \\ V \end{pmatrix} = \begin{pmatrix} U \\ \cdot \end{pmatrix} \tag{4.26}$$

This problem is closely related to the problem of transforming $\binom{A}{V}$ to row echelon form (see [BCS97]).

As in the previous section, the problem can be solved by the cube dag method, using the standard elimination scheme (see e.g. [Mod88, Ort88]). Alternatively, we can compute the decomposition (4.26) by a recursive procedure. This procedure differs from the one described in Section 4.5 in that we cannot compute block inverses. Matrices $\left(\begin{smallmatrix} D & E \\ F & G \end{smallmatrix}\right)$, $\binom{A}{V}$, $\binom{U}{\cdot}$ are partitioned into regular square blocks of size $n/2$,

$$\begin{pmatrix} D_{11} & D_{12} & E_{11} & E_{12} \\ D_{21} & D_{22} & E_{21} & E_{22} \\ F_{11} & F_{12} & G_{11} & G_{12} \\ F_{21} & F_{22} & G_{21} & G_{22} \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ V_{11} & V_{12} \\ \cdot & V_{22} \end{pmatrix} = \begin{pmatrix} U_{11} & U_{12} \\ \cdot & U_{22} \\ \cdot & \cdot \\ \cdot & \cdot \end{pmatrix} \tag{4.27}$$

First the algorithm is applied recursively to $n \times n/2$ matrix $\binom{A_{21}}{V_{11}}$. Taking the decomposition

$$\begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix} \begin{pmatrix} A_{21} \\ V_{11} \end{pmatrix} = \begin{pmatrix} Y_{11} \\ \cdot \end{pmatrix}$$

we obtain

$$\begin{pmatrix} I & \cdot & \cdot & \cdot \\ \cdot & P_{11} & P_{12} & \cdot \\ \cdot & P_{21} & P_{22} & \cdot \\ \cdot & \cdot & \cdot & I \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ V_{11} & V_{12} \\ \cdot & V_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ Y_{11} & Y_{12} \\ \cdot & Y_{22} \\ \cdot & V_{22} \end{pmatrix} \tag{4.28}$$

where

$$\begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix} \begin{pmatrix} A_{22} \\ V_{12} \end{pmatrix} = \begin{pmatrix} Y_{12} \\ Y_{22} \end{pmatrix}$$

In the next stage we apply the algorithm recursively to matrices $\binom{A_{11}}{Y_{11}}$ and $\binom{Y_{22}}{V_{22}}$, obtaining

$$\begin{pmatrix} Q_{11} & Q_{12} & \cdot & \cdot \\ Q_{21} & Q_{22} & \cdot & \cdot \\ \cdot & \cdot & Q_{33} & Q_{34} \\ \cdot & \cdot & Q_{43} & Q_{44} \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ Y_{11} & Y_{12} \\ \cdot & Y_{22} \\ \cdot & V_{22} \end{pmatrix} = \begin{pmatrix} U_{11} & U_{12} \\ \cdot & Z_{12} \\ \cdot & Z_{22} \\ \cdot & \cdot \end{pmatrix} \tag{4.29}$$

Finally, we apply the algorithm recursively to matrix $\binom{Z_{12}}{Z_{22}}$, obtaining

$$\begin{pmatrix} I & \cdot & \cdot & \cdot \\ \cdot & R_{11} & R_{12} & \cdot \\ \cdot & R_{21} & R_{22} & \cdot \\ \cdot & \cdot & \cdot & I \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ \cdot & Z_{12} \\ \cdot & Z_{22} \\ \cdot & \cdot \end{pmatrix} = \begin{pmatrix} U_{11} & U_{12} \\ \cdot & U_{22} \\ \cdot & \cdot \\ \cdot & \cdot \end{pmatrix} \tag{4.30}$$

Matrix $\left(\begin{smallmatrix} D & E \\ F & G \end{smallmatrix}\right)$ can now be computed as the product of the three transformation matrices from (4.28)–(4.30). The base of recursion is the elimination on a 2×1 matrix. If both matrix elements are zero, the decomposition is trivial. Otherwise, an arbitrary nonzero element is used as a pivot. If the other element is also nonzero, it is eliminated by subtracting the pivot multiplied by an appropriate scaling factor.

As opposed to Gaussian elimination without pivoting, the recursive procedure (4.28)–(4.30) is not efficient when implemented in BSP, due to a large number of recursive calls. However, we can improve the efficiency by changing the order in which low-level block operations are executed. We represent the block elimination order defined by (4.28)–(4.30) schematically as

$$\begin{array}{c|c} 2 & 3 \\ 1 & 2 \end{array}$$

Here 1 denotes the elimination of V_{11}, 2 denotes the elimination of Y_{11} and V_{22}, and 3 denotes the elimination of Z_{22}. Vertical bars | represent block multiplication. In this schematic representation, a block elimination can be performed after the block immediately below has been eliminated, and block multiplication immediately to the left has been performed. A block multiplication can be performed after all blocks immediately to the left have been eliminated.

For 4×4 matrices, the elimination scheme of the recursive algorithm is

5	6	8	9
4	5	7	8
2	3	5	6
1	2	4	5

Here the length of the vertical bars corresponds to the size of matrices being multiplied. Note that the elimination 4 in the bottom row has to be computed after the matrix multiplication immediately on the left, and therefore after elimination 3. However, elimination 4 in the left column can be computed immediately after elimination 2, in parallel with elimination 3. In general, we can eliminate any block as soon as the result of multiplication immediately on its left is available. The optimised elimination scheme is

4	5	7	8
3	4	6	7
2	3	5	6
1	2	4	5

Similarly, the original, recursive elimination scheme for 8×8 matrices is

14	15	17	18	23	24	26	27
13	14	16	17	22	23	25	26
11	12	14	15	20	21	23	24
10	11	13	14	19	20	22	23
5	6	8	9	14	15	17	18
4	5	7	8	13	14	16	17
2	3	5	6	11	12	14	15
1	2	4	5	10	11	13	14

and the optimised scheme is

8	9	11	12	16	17	19	20
7	8	10	11	15	16	18	19
6	7	9	10	14	15	17	18
5	6	8	9	13	14	16	17
4	5	7	8	12	13	15	16
3	4	6	7	11	12	14	15
2	3	5	6	10	11	13	14
1	2	4	5	9	10	12	13

This method can be generalised to arbitrary matrix sizes. Decomposition of a $2n \times n$ matrix can be computed by an $r \times r$ optimised elimination scheme, where each entry corresponds to a block of size $n_0 = n/r$. We call such blocks *elementary*. Elimination within an elementary block is performed sequentially by a single BSPRAM processor.

We now describe the allocation of elementary block decomposition tasks and matrix multiplication tasks to the BSPRAM processors. The computation alternates between decomposition of elementary blocks and parallel multiplication of the resulting matrices. Each of these stages is implemented by a superstep. In each decomposition stage, at most one entry from every column of the elimination scheme is computed. In each matrix multiplication stage, at most one matrix multiplication from every column is performed. For any k, we allocate p/k processors to multiplication of matrices of size n/k.

The resulting algorithm is similar to Algorithm 12, in that a real parameter α controls the depth at which block decomposition is performed sequentially. We use an optimised elimination scheme of size $r = p^\alpha$. As before, variation of α results in a tradeoff between the communication and synchronisation costs.

Algorithm 14. *Gaussian elimination with nested block pivoting.*

Parameters: integer $n \geq p^{1+\epsilon}$ for some constant $\epsilon > 0$; real number α, $\alpha_{\min} = \frac{1}{2} + \frac{\log\log p}{2\log p} \leq \alpha \leq \frac{2}{3} + \frac{\log\log p}{\log p} = \alpha_{\max}$.

Input: $n \times n$ matrix A over a finite field.

Output: decomposition $D \cdot A = U$, where D is a full-rank $n \times n$ matrix, and U is an $n \times n$ upper triangular matrix.

Description. The computation is performed on a CRCW BSPRAM(p, g, l). We apply the optimised block elimination procedure to the matrix $\binom{A}{0}$. Denote the matrix size at the current level of recursion by m, keeping n for the original size. Let $n_0 = n/p^\alpha$. Value n_0 is the size of elementary blocks.

We decompose the matrix using an optimised elimination scheme of size $r = p^\alpha$, as described above. Decomposition of elementary blocks is computed sequentially. Matrix multiplication is performed by Algorithm 10.

Cost analysis. The value for $S = S(n)$ can be found from the following recurrence relation:

	$n_0 < m \le n$	$m = n_0$
$S(m) =$	$2 \cdot S(m/2) + O(1)$	$O(1)$

as $S = O(p^\alpha \cdot \log p)$.

Since $r = p^\alpha \le p$, all elementary block decompositions occuring in the same superstep can be performed in parallel. The total cost of elementary block decompositions is

$$\begin{aligned} W_0 &= S_0 \cdot O(n_0^3) = O(n^3 \cdot \log p / p^{2\alpha}) \\ H_0 &= S_0 \cdot O(n_0^2) = O(n^2 \cdot \log p / p^{\alpha}) \\ S_0 &= O(p^\alpha \cdot \log p) \end{aligned}$$

The total computation and communication cost of all matrix multiplications is dominated by the cost of the largest matrix multiplication. Indeed, in each matrix multiplication stage, we compute at most one matrix product of size n, at most two matrix products of size $n/2$, etc. In general, for any k we compute at most k matrix products of size k. Every such product is computed on $p/(2k)$ processors. Let n/K be the size of the largest matrix product occuring in a particular matrix multiplication superstep. The computation cost of this superstep is at most

$$O\Big(\frac{(n/K)^3}{p/K} \sqcup \frac{(n/(2K))^3}{p/(2K)} \sqcup \frac{(n/(4K))^3}{p/(4K)} \sqcup \cdots\Big) = O\Big(\frac{n^3}{K^2 \cdot p}\Big)$$

and the synchronisation cost is at most

$$O\Big(\frac{(n/K)^2}{(p/K)^{2/3}} \sqcup \frac{(n/2K)^2}{(p/2K)^{2/3}} \sqcup \frac{(n/4K)^2}{(p/4K)^{2/3}} \sqcup \cdots\Big) = O\Big(\frac{n^2}{K^{4/3} \cdot p^{2/3}}\Big)$$

where $\sqcup$ denotes the maximum operator. There are at most two supersteps with $K = 1$, at most four supersteps with $K = 2$, etc. In general, there are at most $2K$ supersteps for any particular K. Therefore, the total cost of matrix multiplications is

$$\begin{aligned} W_1 &= O\Big(\frac{n^3}{p} + 2 \cdot \frac{n^3}{2^2 \cdot p} + 4 \cdot \frac{n^3}{4^2 \cdot p} + \cdots\Big) = O(n^3/p) \\ H_1 &= O\Big(\frac{n^2}{p^{2/3}} + 2 \cdot \frac{n^2}{2^{4/3} \cdot p^{2/3}} + 4 \cdot \frac{n^2}{4^{4/3} \cdot p^{2/3}} + \cdots\Big) = O(n^2/p^{2/3}) \\ S_1 &= O(p^\alpha \cdot \log p) \end{aligned}$$

Since $W_0 = O(W_1)$, $H_1 = O(H_0)$, the total BSP cost is

$$W = O(n^3/p) \qquad H = O(n^2 \cdot \log p/p^\alpha) \qquad S = O(p^\alpha \cdot \log p)$$

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

For $\alpha = \alpha_{\min}$, the cost of Algorithm 14 is $W = O(n^3/p)$, $H = O\big(n^2 \cdot (\log p)^{1/2}/p^{1/2}\big)$, $S = O\big(p^{1/2} \cdot (\log p)^{3/2}\big)$. This is slightly higher than the BSP cost of the cube dag method. For $\alpha = \alpha_{\max}$, the cost of Algorithm 14 is $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, $S = O\big(p^{2/3} \cdot (\log p)^2\big)$. In this case the communication cost is as low as in matrix multiplication (Algorithm 10) and Gaussian elimination without pivoting (Algorithm 12). This improvement in communication efficiency is offset by a reduction in synchronisation efficiency. Considerations similar to the ones discussed in Section 4.5 apply to the choice of a particular value of α.

Since the cube dag method is slightly better than Algorithm 14 for $\alpha = \alpha_{\min}$, one might expect that a lower BSP cost may be achieved by a hybrid algorithm, performing elimination by an optimised scheme at the higher level, and decomposing elementary blocks by the cube dag algorithm at the lower level. This would be straightforward if Algorithm 14 were a purely recursive algorithm, similar to Algorithm 12. However, since Algorithm 14 computes the elimination scheme in an optimised non-recursive order, the problem of finding an efficient hybrid algorithm remains open.

To reduce slightly the cost of local computation and communication, one can use fast matrix multiplication instead of standard matrix multiplication for computing block products. In this case the recursion has to be deeper, therefore the synchronisation cost will slightly increase.

Algorithm 14 can be used to compute the QR decomposition of a real matrix. For a 2×1 matrix $\binom{a}{v}$, decomposition (4.26) is given by the Givens rotation

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} a \\ v \end{pmatrix} = \begin{pmatrix} u \\ \cdot \end{pmatrix}$$

where $c = a/(a^2 + v^2)^{1/2}$, $s = v/(a^2 + v^2)^{1/2}$, $u = (a^2 + v^2)^{1/2}$. Recursive equations (4.28)–(4.30) and Algorithm 14 are then directly applied to computation of the QR decomposition of a real matrix by Givens rotations.

4.7 Column pivoting and Householder reflections

In Sections 4.5 and 4.6, we considered Gaussian elimination without pivoting or with nested block pivoting, which is suitable for certain numerical and combinatorial computations on matrices. However, most numerical matrix problems require that in each step, the pivot is chosen globally within

a column (*column pivoting*), or even across the whole matrix (*full pivoting*). For an arbitrary nonsingular real matrix, only full pivoting guarantees numerical stability, and column pivoting is stable on the average. We also considered QR decomposition by Givens rotations, which is similar to nested block pivoting. An alternative method of QR decomposition, Householder reflections, is similar to column pivoting.

We consider Gaussian elimination with column pivoting on rectangular matrices. Let A be an $r \times n$ real matrix, $r \geq n$. Matrix A may not have full rank. The problem consists in finding a full-rank $r \times r$ transformation matrix D, and an $r \times n$ upper triangular matrix U, such that $D \cdot A = U$.

It is easy to obtain a BSPRAM algorithm for Gaussian elimination with column pivoting, if we regard operations on columns (column elimination and matrix-vector multiplication) as elementary operations, which are performed sequentially. In this case the data dependency between elementary tasks is similar to that of triangular system solution. Therefore, we can apply either of the two methods described in Section 4.2: diamond dag algorithm, or recursive substitution. Since an elementary data unit is a column of size r, and both elementary operations have sequential time complexity $O(r)$, the local computation and communication costs of the resulting algorithm are equal to $O(r)$ times the cost of the original triangular system algorithm. For square matrices ($r = n$) this yields $W = O(n^3/p)$, $H = O(n^2)$. The synchronisation cost $S = O(p)$ remains unchanged.

An alternative method is to combine the updates from several column eliminations, using matrix multiplication to perform these updates (see e.g. [Bre91]). Such an approach is often used in parallel numerical software, e.g. in the ScaLAPACK library (see [CDO+96]). Here we give a BSPRAM algorithm based on this approach.

As before, the algorithm can be described recursively. A similar recursive procedure for sequential computation has been introduced in [Tol97]. We partition matrices D, A, U into rectangular blocks

$$\begin{pmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} U_{11} & U_{12} \\ \cdot & U_{22} \end{pmatrix} \tag{4.31}$$

where blocks D_{11}, A_{11}, A_{12}, U_{11}, U_{12} are $n/2 \times n/2$, and the sizes of other blocks conform to the above. First, the algorithm is applied recursively to matrix $\binom{A_{11}}{A_{21}}$. Taking the decomposition

$$\begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix} \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} U_{11} \\ \cdot \end{pmatrix}$$

we obtain

$$\begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} U_{11} & U_{12} \\ \cdot & Y_{22} \end{pmatrix} \tag{4.32}$$

where

$$\begin{pmatrix} P_{11} & P_{12} \\ P_{21} & P_{22} \end{pmatrix} \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix} = \begin{pmatrix} U_{12} \\ Y_{22} \end{pmatrix}$$

Then we apply the algorithm recursively to matrix Y_{22}, obtaining

$$\begin{pmatrix} I & \cdot \\ \cdot & Q_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ \cdot & Y_{22} \end{pmatrix} = \begin{pmatrix} U_{11} & U_{12} \\ \cdot & U_{22} \end{pmatrix} \tag{4.33}$$

Matrix D can now be computed as the product of the two transformation matrices from (4.32)–(4.33). The base of recursion is the elimination on a single column, which involves the search of the largest column element as a pivot, and elimination of all other elements by subtracting the pivot multiplied by an appropriate scaling factor.

We now describe the allocation of block decomposition tasks and block multiplication tasks in (4.32)–(4.33) to the BSPRAM processors. Initially, all p processors are available to compute the decomposition $D \cdot A = U$. There is no substantial parallelism between block decomposition and block multiplication tasks in (4.32)–(4.33); we can only exploit the parallelism within block multiplication. Therefore, the recursion tree has to be computed in depth-first order. In each level of recursion, every block multiplication in (4.32)–(4.33) is performed in parallel by all processors available at that level. Each block decomposition is also performed in parallel by all processors available at that level, if the block size is large enough. When blocks become sufficiently small, block decomposition is computed sequentially by an arbitrarily chosen processor.

The resulting algorithm is as follows.

Algorithm 15. *Gaussian elimination with column pivoting.*

Parameter: integer $n \geq p$.

Input: $r \times n$ real matrix A, $r \geq n$.

Output: decomposition $D \cdot A = U$, where D is a full-rank $r \times r$ matrix, and U is an $r \times n$ upper triangular matrix.

Description. The computation is performed on a CRCW BSPRAM(p, g, l), and is defined by recursion on the size of the matrices. Denote the matrix width at the current level of recursion by m, keeping n for the original width. Let $n_0 = n/p$. Value n_0 is the threshold, at which the algorithm switches from parallel to sequential computation.

In each level of recursion, the matrices are divided as shown in (4.31). Then, computation (4.32)–(4.33) is performed by the following schedule.

Small blocks. If $1 \leq m \leq n_0$, choose an arbitrary processor from all currently available, and compute (4.32)–(4.33) on that processor.

Large blocks. If $n_0 < m \leq n$, compute U_{11}, U_{12} by recursion. Then compute Y_{22} by Algorithm 10. Finally, compute U_{22} by recursion. Each of these computations is performed with all available processors. The result is the decomposition (4.31).

Cost analysis. The value for $S = S(n)$ can be found from the following recurrence relation:

	$n_0 < m \leq n$	$m = n_0$
$S(m) =$	$2 \cdot S(m/2) + O(1)$	$O(1)$

as $S = O(p)$.

The local computation cost W and communication cost H are dominated by the decomposition of elementary blocks: $W = O(p) \cdot O(r^2 n_0)$, $H = O(p) \cdot O(r n_0)$. For square matrices ($r = n$) we obtain:

$$W = O(n^3/p) \qquad H = O(n^2) \qquad S = O(p)$$

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

The above analysis shows that the recursive algorithm for column pivoting has the same BSP cost as the column-based triangular system algorithm. Using block multiplication for matrix updates does not improve the asymptotic communication cost of column pivoting. A straightforward extension of the method is *block column pivoting*, where pivot search is performed locally in a rectangular block of columns, instead of a single column, thus improving numerical properties of the algorithm.

In contrast with previous pivoting methods, using fast matrix multiplication in column pivoting will reduce slightly the cost of local computation, but the communication cost will remain unchanged. As before, the recursion will have to be deeper, therefore the synchronisation cost will slightly increase.

Algorithm 15 can be used to compute the QR decomposition of a real matrix by Householder reflections. In this method, a column u is eliminated by a symmetric orthogonal transformation matrix $P = I - 2v \cdot v^T$, where v is a unit vector constructed from column u in linear time. Recursive equations (4.32)–(4.33) and Algorithm 15 are then directly applied to computation of the QR decomposition of a real matrix by Householder reflections.

Finally, we consider Gaussian elimination with full pivoting. In this method, pivot search is done across the whole matrix on each elimination stage. For real matrices, full pivoting guarantees numerical stability; however, in practice it is used less frequently than column pivoting, due to a higher computation cost.

In contrast with matrix problems considered previously, there is no known block method for Gaussian elimination with full pivoting. The only

viable approach to this problem seems to be fine-grain. Consider Gaussian elimination with full pivoting on an $n \times n$ matrix. The matrix is partitioned across the processors by regular square blocks of size $n/p^{1/2}$. Each of n elimination stages is implemented by $O(1)$ supersteps. The cost of each stage is dominated by the cost of a rank-one matrix update: $w = O(n^2/p)$, $h = O(n/p^{1/2})$. The total BSP cost of the algorithm is therefore $W = O(n) \cdot w = O(n^3/p)$, $H = O(n) \cdot h = O(n^2/p^{1/2})$, $S = O(n)$. Note that the communication cost H is lower than that of Algorithm 15, but the synchronisation cost S is much higher. The described fine-grain algorithm can be applied to Gaussian elimination with column pivoting, in which case it presents a discontinuous communication-synchronisation tradeoff with Algorithm 15.

Chapter 5

Graph computation in the BSP model

5.1 Fast Boolean matrix multiplication

Graph computation is a large and particularly well-studied area of combinatorial computation. It has a strong connection to matrix computation, since a graph can be represented by its adjacency matrix. Many graph algorithms have matrix analogues, and vice versa.

In the simplest case of an unweighted graph, the adjacency matrix is Boolean. In this section, we consider the problem of multiplying two $n \times n$ Boolean matrices, using conjunction $\wedge$ and disjunction $\vee$ as multiplication and addition respectively. The straightforward method is standard matrix multiplication, of sequential complexity $\Theta(n^3)$. There are also subcubic methods, including Kronrod's algorithm (also known as Four Russians' algorithm, see e.g. [AHU76]), and a recent algorithm from [BKM95]. The lowest known exponent is achieved by fast Strassen-type multiplication. In this method, the Boolean matrices are viewed as $(0,1)$-matrices over the ring of integers modulo $n+1$ (see e.g. [Pat74, AHU76], [LD90, pages 537–538], or [CLR90, pages 747–748]). As shown in Section 4.4, the fast matrix multiplication algorithm has BSP cost $W = O(n^\omega/p)$, $H = O(n^2/p^{2/\omega})$, $S = O(1)$. For matrices over a general semiring, these cost values are independently optimal (Theorem 8). It is natural to ask whether the asymptotic communication cost H can be reduced by using properties specific to Boolean matrices.

In this section we give a positive answer to this question, describing an algorithm with communication cost $H = O(n^2/p)$. The proposed algorithm is not practical, since it works only for astronomically large matrices, and involves huge constant factors. However, the method is of theoretical importance, because it indicates that the lower bound $H = \Omega(n^2/p^{2/\omega})$, which is easy to prove for a general semiring, cannot be extended to the Boolean

case.

We first give an intuitive explanation of the method. The main idea is to find the high-level structure of the two matrices and of their product. As soon as this basic structure is determined, the full structure can be found with little extra communication. In contrast with the standard and Strassen-type methods, the resulting algorithm is non-oblivious.

Let us consider the standard $\Theta(n^3)$ computation of the product $A \wedge B = C$, where A, B, C are square Boolean matrices of size n. As in Section 4.3, we represent the n^3 computed Boolean products by a cube of volume n^3 in integer three-dimensional space. The proposed algorithm performs communication only on matrices containing few ones, or few zeros. A matrix with m ones (or m zeros) can be communicated by sending the m indices of the ones (respectively, the zeros). If A contains at most n^2/p ones, the multiplication problem can be solved on a BSPRAM in $W = O(n^3/p)$, $H = O(n^2/p)$, $S = O(1)$ by partitioning the cube into layers of size $n/p \times n \times n$, parallel to the coordinate plane representing matrix A. Symmetrically, if B contains at most n^2/p ones, the problem can be solved at the same cost by a similar partitioning into layers of size $n \times n/p \times n$, parallel to the coordinate plane representing matrix B. If both A and B are dense, we partition the cube into layers of size $n \times n \times n/p$, parallel to the coordinate plane representing matrix C. Assuming for the moment that A and B are random matrices, it is likely that the partial product computed by each of the layers contains at most n^2/p zeros. Again, the problem can be solved in $W = O(n^3/p)$, $H = O(n^2/p)$, $S = O(1)$. The remaining case is when A and B have relatively many ones, but C still has relatively many zeros. We argue that in this case the triple A, B, C must have a special structure that allows us to decompose the computation into three matrix products corresponding to the three easy cases above. Computation of this structure requires no prior knowledge of C. Its cost is polynomial in n (more precisely, of order n^ω), but exponential in p.

The structure of a Boolean matrix product is best described in the language of graph theory. To expose the symmetry of the problem, we modify it as follows. For a Boolean matrix X, let $\overline{X}$ denote the Boolean complement to the transpose of X, i.e. $\overline{X}[i,j] = \overline{X[j,i]}$. We replace C by $\overline{C}$, and look for a C such that $A \wedge B = \overline{C}$. Boolean matrices A, B, C define a tripartite graph, if we consider them as adjacency matrices of its three bipartite connection subgraphs. We will denote this tripartite graph by $G = (A, B, C)$. Since A, B, C are square $n \times n$ matrices, the graph G is *equitripartite*, with the size of each partite class equal to n.

A simple undirected graph is called *triangle-free* if it does not contain a triangle (a cycle of length three). The graph G is triangle-free — existence of a triangle in G would imply that for some i, j, k, $A[i,j] = B[j,k] = C[k,i] = 1$, therefore $A[i,j] \wedge B[j,k] = 1$, but $\overline{C}[i,k] = \overline{C[k,i]} = \overline{1} = 0$. Note that the property of a graph G to be triangle-free is symmetric: matrix C does not

play any special role compared to A and B.

The following simple result is not necessary for the derivation of our algorithm, and is given only to illustrate the connection between Boolean matrix multiplication and triangle-free graphs. Let us call the factors A and B *maximal*, if changing a zero to a one in any position of A or B results in changing some zeros to ones in the product $\overline{C}$ (and therefore some ones to zeros in C). A triangle-free graph is called *maximal* if the addition of any new edge creates a triangle. When we consider tripartite triangle-free graphs, we call such a graph *maximal* if we cannot add any new edge so that the resulting graph is still tripartite and triangle-free. Note that by this definition, a maximal tripartite triangle-free graph may not be maximal as a general triangle-free graph. We have the following lemma.

Lemma 2. *Let A, B, C be arbitrary $n \times n$ Boolean matrices. The following statements are equivalent:*

(i) *A, B are maximal matrices such that $A \wedge B = \overline{C}$;*

(ii) *$A \wedge B = \overline{C}$, $B \wedge C = \overline{A}$, $C \wedge A = \overline{B}$;*

(iii) *$G = (A, B, C)$ is a maximal equitripartite triangle-free graph.*

Proof. Straightforward application of the definitions. ■

Lemma 2 shows that in the product $A \wedge B = \overline{C}$, the matrices A, B, C are in a certain sense interchangeable: any matrix which is a Boolean product can also be characterised as a maximal Boolean factor. It also gives a characterisation of such matrices by maximal triangle-free graphs.

One of the few references to maximal equitripartite triangle-free graphs appears in [Bol78]. In particular, [Bol78, pages 324–325] states the problem of finding the minimum possible density of such a graph; it is easy to see from the discussion above that this problem is closely related to Boolean matrix multiplication. It is then noted in [Bol78] that, as of the time of writing, the minimum density problem was "completely unresolved". Since then, however, a general approach to problems of this kind has been developed. The basis of this approach is Szemerédi's Regularity Lemma (see e.g. [KS96]). Here we apply this lemma directly to the Boolean matrix multiplication problem; it might also be applicable to similar extremal graph problems, including the minimum density problem.

In the definitions and theorems below, we follow [KS96, Die97]. Let $G = (V, E)$ be a simple undirected graph. Let $v(G) = |V|$, $e(G) = |E|$. For disjoint $X, Y \subset V$, let $e(X, Y)$ denote the number of edges between X and Y. We define the *density* of the bipartite subgraph (X, Y) as $d(X, Y) = e(X, Y)/(|X| \cdot |Y|)$. For disjoint $A, B \subset V$ we call (A, B) an *ϵ-regular subgraph*, if for every $X \subset A$, $|X| > \epsilon|A|$, and $Y \subset B$, $|Y| > \epsilon|B|$, we have $|d(X, Y) - d(A, B)| < \epsilon$. We say that G has an *(ϵ, d)-partitioning of*

size m, if V can be partitioned into m disjoint subsets of equal size, called *clusters*, such that for any two clusters A, B, the bipartite subgraph (A, B) is either ϵ-regular with density at least d, or empty. A *cluster graph* of an (ϵ, d)-partitioning is a graph with m nodes corresponding to the clusters, in which two nodes are connected by an edge if and only if the two corresponding clusters form a non-empty bipartite subgraph. If G is an equitripartite graph, then each cluster is a subset of one of the three parts, and the cluster graph is also equitripartite.

We will not apply the definition of ϵ-regularity directly. Instead, we will use the following theorem (in a slightly more general form, paper [KS96] calls it the Key Lemma; a further generalisation is known as the Blow-Up Lemma).

Theorem 9 (Komlós, Simonovits). *Let $d > \epsilon > 0$. Let G be a graph with an (ϵ, d)-partitioning, and let R be the cluster graph of this partitioning. Let H be a subgraph of R with maximum degree $\Delta > 0$. If $\epsilon \leq (d-\epsilon)^{\Delta}/(2+\Delta)$, then G contains a subgraph isomorphic to H.*

Proof. See [KS96, Die97]. ■

Since we are interested in triangle-free graphs, we take H to be a triangle. By simplifying the condition on d and ϵ, we obtain the following corollary of Theorem 9: if $d \leq 4/5$, $\epsilon \leq d^2/4$, and G is triangle-free, then its cluster graph R is also triangle-free.

Our main tool is Szemerédi's Regularity Lemma. Informally, it states that any graph can be transformed into a graph with an (ϵ, d)-partitioning by removing a small number of nodes and edges. Its precise statement, slightly adapted from [KS96], is as follows.

Theorem 10 (Szemerédi). *For every $\epsilon > 0$ there is an $M = M(\epsilon)$ such that for any d, $\epsilon \leq d \leq 1$, an arbitrary graph G contains a subgraph G_0 with an (ϵ, d)-partitioning of size at most M, and $e(G \setminus G_0) \leq (d+\epsilon)\big(v(G)\big)^2$.*

Proof. See [KS96, Die97]. ■

Note that if $e(G) = o\big(v(G)^2\big)$, the statement of Theorem 10 becomes trivial. Also note that for any $d \geq \epsilon$, an (ϵ, d)-partitioning can be obtained from an (ϵ, ϵ)-partitioning by simply excluding the ϵ-regular subgraphs with densities between ϵ and d.

For an equitripartite graph G of size $3n$, where $n \geq 2^{2^{12}\epsilon^{-18}}$, paper [ADL+94] gives an algorithm which finds the subgraph G_0 in sequential time $O\big(2^{2^{10}\epsilon^{-17}} \cdot n^{\omega}\big)$, where ω is the exponent of matrix multiplication (currently 2.376 by [CW90]). The size of the resulting (ϵ, d)-partitioning is at most $M = 2^{2^{10}\epsilon^{-17}}$.

We are now able to describe the proposed communication-efficient algorithm for computing $A \wedge B = \overline{C}$, where A, B, C are Boolean matrices of size

n. Let $G = (A, B, C)$. We represent the n^3 elementary Boolean products as a cube of volume n^3 in the three-dimensional index space. The cube G is partitioned into p^3 regular cubic blocks of size n/p. Each block is local to a particular processor, and corresponds to an equitripartite triangle-free graph[1] $\mathsf{G} = (\mathsf{A}, \mathsf{B}, \mathsf{C})$, where $\mathsf{A} = A[[I, J]]$, $\mathsf{B} = B[[J, K]]$ for some $1 \le I, J, K \le p$, and $\overline{\mathsf{C}} = \mathsf{A} \wedge \mathsf{B}$. We shall identify the cubic block with its graph G.

Let us choose positive real numbers ϵ and d, such that $\epsilon \le d^2/4$ (necessary as a condition of Theorem 9), and $d + \epsilon \le 2/p^2$ (necessary to ensure that the communication cost $H = O(n^2/p)$ after the application of Theorem 10). We take $d = 1/p^2$, $\epsilon = d^2/4 = 1/(4p^4)$. By Theorem 10, we can find a large subgraph $\mathsf{G}_0 \subset \mathsf{G}$ with an (ϵ, d)-partitioning of size $3m \le M(\epsilon)$. Let $\mathsf{G}_0 = (\mathsf{A}_0, \mathsf{B}_0, \mathsf{C}_0)$. Also let $\Delta\mathsf{G} = \mathsf{G} \setminus \mathsf{G}_0$, and $\Delta\mathsf{G} = (\Delta\mathsf{A}, \Delta\mathsf{B}, \Delta\mathsf{C})$. We have $e(\Delta\mathsf{G}) \le (d + \epsilon)n^2/p^2$. Note that $\mathsf{A} = \mathsf{A}_0 \vee \Delta\mathsf{A}$, $\mathsf{B} = \mathsf{B}_0 \vee \Delta\mathsf{B}$, $\mathsf{C} = \mathsf{C}_0 \vee \Delta\mathsf{C}$, and $\overline{\mathsf{C}} = \mathsf{A} \wedge \mathsf{B} = \overline{\mathsf{C}'_0} \vee \overline{\Delta\mathsf{C}'}$, where $\overline{\mathsf{C}'_0} = \mathsf{A}_0 \wedge \mathsf{B}_0$, and $\overline{\Delta\mathsf{C}'} = (\Delta\mathsf{A} \wedge \mathsf{B}) \vee (\mathsf{A} \wedge \Delta\mathsf{B})$.

The (ϵ, d)-partitioning of the graph G_0 is, up to a permutation of indices, a partitioning of G_0 into m^3 regular cubic subblocks. Let us denote a subblock of G_0 by $\mathsf{G}_0[[i, j, k]] = (\mathsf{A}_0[[i, j]], \mathsf{B}_0[[j, k]], \mathsf{C}_0[[k, i]])$, $0 \le i, j, k < m$. Let $\mathsf{G}[[i, j, k]]$ and $\Delta\mathsf{G}[[i, j, k]]$ denote similar subblocks of G and $\Delta\mathsf{G}$.

Consider an arbitrary subblock $\mathsf{C}_0[[k, i]]$. If $\mathsf{C}_0[[k, i]]$ is a zero matrix, then $\mathsf{C}[[k, i]] = \Delta\mathsf{C}[[k, i]]$, and $\overline{\mathsf{C}[[k, i]]} = \overline{\Delta\mathsf{C}[[k, i]]}$. If $\mathsf{C}_0[[k, i]]$ is non-zero, then for any j, $0 \le j < m$, either $\mathsf{A}_0[[i, j]]$ or $\mathsf{B}_0[[j, k]]$ is a zero matrix by Theorem 9. Therefore, $\overline{\mathsf{C}'_0[[k, i]]}$ is a zero matrix, and $\overline{\mathsf{C}[[k, i]]} = \overline{\Delta\mathsf{C}'[[k, i]]}$. The two cases ($\mathsf{C}_0[[k, i]]$ zero or non-zero) can be distinguished by the cluster graph of G_0 alone. The product $\mathsf{A} \wedge \mathsf{B} = \overline{\mathsf{C}}$ can therefore be found by selecting each subblock of $\overline{\mathsf{C}}$ from $\overline{\Delta\mathsf{C}}$ or $\overline{\Delta\mathsf{C}'} = (\Delta\mathsf{A} \wedge \mathsf{B}) \vee (\mathsf{A} \wedge \Delta\mathsf{B})$, where the choice is governed by the cluster graph. The matrix $\overline{\mathsf{C}'_0} = \mathsf{A}_0 \wedge \mathsf{B}_0$ need not be computed.

In order to compute the (ϵ, d)-partitioning for all p^3 blocks of G at the communication cost $H = O(n^2/p)$, blocks must be grouped into p layers of size $n \times n \times n/p$, with p^2 blocks in each layer. The computation of the two Boolean matrix products in $\overline{\Delta\mathsf{C}'} = (\Delta\mathsf{A} \wedge \mathsf{B}) \vee (\mathsf{A} \wedge \Delta\mathsf{B})$ requires that the same p^3 cubic blocks are divided into p similar layers of sizes $n/p \times n \times n$ and $n \times n/p \times n$, respectively.

The number of nodes in the cluster graph of each block is $O\left(2^{2^{10}\epsilon^{-17}}\right) = O\left(2^{2^{44}p^{68}}\right)$. Therefore, each cluster graph contains $O\left(2^{2^{45}p^{68}}\right)$ edges. Each processor computes cluster graphs for p^2 blocks, therefore the total number of cluster graph edges computed by a processor is at most $O\left(2^{2^{46}p^{68}}\right)$. The cluster graphs must be exchanged between the processors. To obtain an algorithm with low communication cost, we require that this number of edges is at most $O(n^2/p)$, therefore it is sufficient to assume that $n = \Omega\left(2^{2^{45}p^{68}}\right)$.

The algorithm is as follows.

[1] We use the sans-serif font for blocks, in order to reduce the number of subscripts.

Algorithm 16. *Boolean matrix multiplication.*

Parameter: integer $n = \Omega\left(2^{2^{45}p^{68}}\right)$.

Input: $n \times n$ Boolean matrices A, B.

Output: $n \times n$ Boolean matrix C, such that $A \wedge B = \overline{C}$.

Description. The algorithm is performed on a CRCW BSPRAM(p, g, l). The Boolean matrix product $A \wedge B = \overline{C}$ is represented as a cube of size n in integer three-dimensional space. This cube is partitioned into p^3 regular cubic blocks.

The algorithm proceeds in three stages. We use the notation of Theorem 10 and of the subsequent discussion.

First stage. Matrix $\overline{C}$ is initialised with zeros. The cube is partitioned into layers of size $n \times n \times n/p$, each containing p^2 cubic blocks. Every processor picks a layer, reads the necessary blocks of matrices A, B, and for each cubic block G computes the graph G_0 and the decompositions $\mathsf{A} = \mathsf{A}_0 \vee \Delta\mathsf{A}$, $\mathsf{B} = \mathsf{B}_0 \vee \Delta\mathsf{B}$, $\mathsf{C} = \mathsf{C}_0 \vee \Delta\mathsf{C}$. Then for each block G the processor writes back the matrices $\Delta\mathsf{A}$, $\Delta\mathsf{B}$, $\Delta\mathsf{C}$, and the cluster graph of G.

Second stage. The Boolean products $\Delta\mathsf{A} \wedge \mathsf{B}$ and $\mathsf{A} \wedge \Delta\mathsf{B}$ are computed by partitioning the cube into layers of size $n/p \times n \times n$ and $n \times n/p \times n$, respectively, each containing p^2 cubic blocks. Then the Boolean sum $\overline{\Delta\mathsf{C}'} = (\Delta\mathsf{A} \wedge \mathsf{B}) \vee (\mathsf{A} \wedge \Delta\mathsf{B})$ is computed.

Third stage. The blocks of matrix $\overline{C}$ are partitioned equally among the processors. Every processor reads the necessary cluster graphs, and then computes each block $\overline{\mathsf{C}}$ by selecting its subblocks from $\overline{\Delta\mathsf{C}}$ or $\overline{\Delta\mathsf{C}'}$, as directed by the cluster graph.

The resulting matrix $\overline{C}$ is the Boolean matrix product of A and B.

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

$$W = O\left(2^{2^{46}p^{68}} \cdot n^{\omega}\right) \qquad H = O(n^2/p) \qquad S = O(1)$$

Hence, for any $\gamma > \omega$, and $n = \Omega\left(2^{2^{46\cdot(\gamma-\omega)^{-1}}\cdot p^{68}}\right)$, the local computation cost is $W = O(n^{\gamma}/p)$, and still $H = O(n^2/p)$, $S = O(1)$. Although the algorithm is essentially non-oblivious, it is communication-oblivious. Its slackness and granularity are $\sigma = n^2/p$, $\gamma = 1$. ■

Algorithm 16 is asymptotically optimal in communication, since examining the input already costs $H = \Omega(n^2/p)$. Moreover, its local computation cost is polynomial in n with exponent ω. Therefore, for sufficiently large n, the algorithm improves on the asymptotic cost of any Strassen-type BSP algorithm with exponent ψ, $\omega < \psi < 3$. The algorithm is trivially optimal in synchronisation.

5.2 Algebraic path computation

In this section we consider the problem of finding the closure of a square matrix over a semiring. This problem is also known as the *algebraic path problem*. It unifies many seemingly unrelated computational problems, such as graph connectivity, network reliability, regular language generation, network capacity. All these tasks can be viewed as instances of the algebraic path problem for an appropriately chosen semiring. More information on applications of the algebraic path problem can be found in [Car79, Zim81, GM84a, GM84b, Rot90].

Let an $n \times n$ matrix A over a semiring represent a *weighted graph* with nodes $1, \ldots, n$. The *length* of an edge $i \to j$ is defined as the semiring element $A[i, j]$. If the graph is not complete, we assume that non-edges have length zero. Let $A^* = I + A + A^2 + \cdots$ be the closure of matrix A (it is not guaranteed to exist in a general semiring). The *distance* between nodes i, j is defined as the semiring element $A^*[i, j]$. Note that in this general setting, the distance does not have to correspond to any particular "shortest" path in the graph. In the special case where the semiring is the set of all nonnegative real numbers with ∞, and the operations min and $+$ are used as semiring addition and multiplication respectively, lengths and distances have their standard graph-theoretic meaning — in particular, ∞ plays the role of the zero, and the distances are realised by shortest paths. We will return to this special case in Section 5.4.

In order to compute the closure of a square matrix over a general semiring, we use Gaussian elimination without pivoting. The method is similar to the one described in Section 4.5. In the absence of pivoting, Gaussian elimination over a general semiring is not guaranteed to terminate. Guaranteed termination can be achieved by restricting the domain (e.g. considering closed semirings instead of arbitrary semirings), or by restricting the type of the matrix, as we did in Section 4.5 for numerical matrices with certain special properties. In the case of numerical matrices, computation of the matrix closure corresponds to matrix inversion: $A^* = (I - A)^{-1}$.

Let A be an $n \times n$ matrix over a semiring. We assume that the closure of a semiring element can be computed in time $O(1)$, whenever this closure exists. Matrix closure A^* can be computed by sequential Gaussian elimination in time $\Theta(n^3)$, provided that the computation terminates. This method is asymptotically optimal for matrices over a general semiring, which can be shown by a standard reduction of the matrix multiplication problem.

The BSP matrix closure computation is similar to LU decomposition (see Section 4.5). The problem can be solved by the cube dag method, giving BSP cost $W = O(n^3/p)$, $H = O(n^2/p^{1/2})$, $S = O(p^{1/2})$, or by recursive block Gauss–Jordan elimination. We repeat the description of block Gauss–Jordan elimination from Section 4.5, making the changes necessary to adapt it to the algebraic path problem.

For convenience we assume that the resulting matrix A^* must replace the original matrix A in the main memory of BSPRAM. The algorithm works by dividing the matrix into square blocks of size $n/2$,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \tag{5.1}$$

and then applying block Gauss–Jordan elimination:

$$\begin{aligned} \bar{A}_{11} &\leftarrow A_{11}^* & \bar{\bar{A}}_{22} &\leftarrow \bar{A}_{22}^* \\ \bar{A}_{12} &\leftarrow \bar{A}_{11} A_{12} & \bar{\bar{A}}_{21} &\leftarrow \bar{\bar{A}}_{22} \bar{A}_{21} \\ \bar{A}_{21} &\leftarrow A_{21} \bar{A}_{11} & \bar{\bar{A}}_{12} &\leftarrow \bar{A}_{12} \bar{\bar{A}}_{22} \\ \bar{A}_{22} &\leftarrow A_{22} + A_{21} \bar{A}_{11} A_{12} & \bar{\bar{A}}_{11} &\leftarrow \bar{A}_{11} + \bar{A}_{21} \bar{\bar{A}}_{22} \bar{A}_{12} \end{aligned} \tag{5.2}$$

after which every $\bar{\bar{A}}_{ij}$ overwrites A_{ij}. The procedure can be applied recursively to find A_{11}^* and $\bar{A}_{22}^*$. The resulting matrix is

$$A^* = \begin{pmatrix} A_{11}^* + A_{11}^* A_{12} \cdot G^* \cdot A_{21} A_{11}^* & A_{11}^* A_{12} \cdot G^* \\ G^* \cdot A_{21} A_{11}^* & G^* \end{pmatrix} \tag{5.3}$$

where $G = A_{22} + A_{21} A_{11}^* A_{12}$. The computation terminates, if all taken closures exist.

We now describe the allocation of block closure tasks and block multiplication tasks in (5.2) to the BSPRAM processors. Initially, all p processors are available to compute the closure A^*. There is no substantial parallelism between block closure and block multiplication tasks in (5.2); we can only exploit the parallelism within block multiplication. Therefore, the recursion tree is computed in depth-first order. In each level of recursion, every block multiplication in (5.2) is performed in parallel by all processors available at that level. Each block closure in (5.2) is also performed in parallel by all processors available at that level, if the block size is large enough. When blocks become sufficiently small, block closure is computed sequentially by an arbitrarily chosen processor.

The depth at which the algorithm switches from p-processor to single-processor computation can be varied. This variation allows us to trade off the costs of communication and synchronisation in a certain range. In order to account for this tradeoff, we introduce a real parameter α, controlling the depth of parallel recursion. The algorithm is as follows.

Algorithm 17. *Algebraic path computation.*

Parameters: integer $n \geq p$; real number α, $\alpha_{\min} = 1/2 \leq \alpha \leq 2/3 = \alpha_{\max}$.

Input: $n \times n$ matrix A over a semiring.

Output: $n \times n$ matrix closure A^* (assuming it exists), overwriting A.

Description. The computation is performed on a CRCW BSPRAM(p, g, l), and is defined by recursion on the size of the matrix. Denote the matrix size at the current level of recursion by m, keeping n for the original size. Let $n_0 = n/p^\alpha$. Value n_0 is the threshold, at which the algorithm switches from parallel to sequential computation.

In each level of recursion, the matrix is divided into regular square blocks of size $m/2$ as shown in (5.1). Then, computation (5.2) is performed by the following schedule.

Small blocks. If $1 \le m \le n_0$, choose an arbitrary processor from all currently available, and compute (5.2) on that processor.

Large blocks. If $n_0 < m \le n$, compute $\bar{A}_{11}$ by recursion. Then compute $\bar{A}_{12}$, $\bar{A}_{21}$, $\bar{A}_{22}$ by Algorithm 10. After that, compute $\bar{\bar{A}}_{22}$ by recursion. Finally, compute $\bar{\bar{A}}_{21}$, $\bar{\bar{A}}_{12}$, $\bar{\bar{A}}_{11}$ by Algorithm 10. Each of these computations is performed with all available processors. Overwrite every A_{ij} by $\bar{\bar{A}}_{ij}$, obtaining the matrix closure (5.3).

Cost analysis. Recurrence relations, identical to the ones used for Algorithm 12, give the BSP cost

$$W = O(n^3/p) \qquad H = O(n^2/p^\alpha) \qquad S = O(p^\alpha)$$

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

Similarly to Algorithm 12, Algorithm 17 with $\alpha = \alpha_{\max} = 2/3$ is better suited for large values of n, and $\alpha = \alpha_{\min} = 1/2$ may perform better for a moderate n.

Lower bounds for the BSP cost of algebraic path computation are also similar to the bounds given in Section 4.5. In particular, the problem of computing the $n \times n$ matrix product $A \cdot B$ can be reduced to algebraic path computation by considering the closure of a $3n \times 3n$ lower triangular matrix

$$\begin{pmatrix} I & & \\ B & I & \\ \cdot & A & I \end{pmatrix}^* = \begin{pmatrix} I & & \\ B & I & \\ A \cdot B & A & I \end{pmatrix} \tag{5.4}$$

(see e.g. [Pat74, CLR90]). Therefore, the lower bound $H = \Omega(n^2/p^{2/3})$ for standard matrix multiplication (Theorem 7) holds also for matrix closure over a general semiring.

If the ground semiring is a commutative ring with unit, fast matrix multiplication can be used instead of standard matrix multiplication for computing block products. The modified algorithm is as follows.

Algorithm 18. *Fast algebraic path computation.*

Parameters: integer $n \ge p^{3/\omega}$; real number α, $\alpha_{\min} = 1/(\omega - 1) \le \alpha \le 2/\omega = \alpha_{\max}$.

Input: $n \times n$ matrix A over a commutative ring with unit.

Output: $n \times n$ matrix closure A^* (assuming it exists), overwriting A.

Description. The computation is identical to Algorithm 17, except that block multiplication is performed by Algorithm 11, rather than Algorithm 10.

Cost analysis. Recurrence relations, identical to the ones used for Algorithm 13, give the BSP cost

$$W = O(n^\omega/p) \qquad H = O(n^2/p^\alpha) \qquad S = O(p^\alpha)$$

The algorithm is oblivious, with slackness and granularity $\sigma = \gamma = n^2/p^{2\omega^{-1}}$. ■

If A is a real symmetric positive definite matrix, then the algorithm for computing the inverse A^{-1} can be obtained by transforming equations (5.2) into the form

$$\begin{aligned} \bar{A}_{11} &\leftarrow A_{11}^{-1} & \bar{\bar{A}}_{22} &\leftarrow \bar{A}_{22}^{-1} \\ \bar{A}_{12} &\leftarrow \bar{A}_{11}A_{12} & \bar{\bar{A}}_{21} &\leftarrow -\bar{\bar{A}}_{22}\bar{A}_{21} \\ \bar{A}_{21} &\leftarrow A_{21}\bar{A}_{11} & \bar{\bar{A}}_{12} &\leftarrow -\bar{A}_{12}\bar{\bar{A}}_{22} \\ \bar{A}_{22} &\leftarrow A_{22} - A_{21}\bar{A}_{11}A_{12} & \bar{\bar{A}}_{11} &\leftarrow \bar{A}_{11} + \bar{A}_{21}\bar{\bar{A}}_{22}\bar{A}_{12} \end{aligned} \tag{5.5}$$

after which every $\bar{\bar{A}}_{ij}$ overwrites A_{ij}. It is easy to see that matrices A_{11}, $\bar{A}_{22}$ are symmetric positive definite, therefore the procedure can be applied recursively to find A_{11}^{-1} and $\bar{A}_{22}^{-1}$. The resulting matrix is

$$A^{-1} = \begin{pmatrix} A_{11}^{-1} + A_{11}^{-1}A_{12} \cdot G^{-1} \cdot A_{21}A_{11}^{-1} & -A_{11}^{-1}A_{12} \cdot G^{-1} \\ -G^{-1} \cdot A_{21}A_{11}^{-1} & G^{-1} \end{pmatrix} \tag{5.6}$$

where $G = A_{22} - A_{21}A_{11}^{-1}A_{12}$. Thus, matrix inversion can be performed by a standard algorithm similar to Algorithms 12, 17, or by a fast algorithm similar to Algorithms 13, 18.

5.3 Algebraic paths in acyclic graphs

In the previous section we considered the algebraic path problem on an arbitrary directed graph. It is interesting to analyse the same problem in the special case where the underlying graph is acyclic.

A directed acyclic graph can be topologically sorted by an all-pairs shortest paths algorithm with BSP cost $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, $S = O(\log p)$ (see Section 5.4). From now on, we assume that the input graph is topologically sorted, so that the edges are directed from higher-indexed to lower-indexed nodes. Such a graph is represented by a lower triangular matrix.

Let A be an $n \times n$ lower triangular matrix over a semiring. As before, we solve the problem of computing the matrix closure A^* by recursive block Gauss–Jordan elimination. We assume that the resulting lower triangular matrix A^* must replace the original matrix A in the main memory of BSPRAM. The algorithm works by dividing the matrix into square blocks of size $n/2$,

$$A = \begin{pmatrix} A_{11} & \\ A_{21} & A_{22} \end{pmatrix} \tag{5.7}$$

and then applying block Gauss–Jordan elimination:

$$\begin{aligned} \bar{A}_{11} &\leftarrow A_{11}^* \\ \bar{A}_{22} &\leftarrow A_{22}^* \\ \bar{A}_{21} &\leftarrow \bar{A}_{22} A_{21} \bar{A}_{11} \end{aligned} \tag{5.8}$$

after which every $\bar{A}_{ij}$ overwrites A_{ij}. The procedure can be applied recursively to find A_{11}^* and A_{22}^*. The resulting matrix is

$$A^* = \begin{pmatrix} A_{11}^* & \\ A_{22}^* A_{21} A_{11}^* & A_{22}^* \end{pmatrix} \tag{5.9}$$

The computation terminates, if all taken closures exist.

An important difference from the case of arbitrary matrices is that the block closures A_{11}^* and A_{22}^* in (5.8) can be computed independently and simultaneously. In order to exploit this feature, we partition the set of available processors into two subsets of equal size, each computing one of the two block closures. This partitioning takes place in every level of recursion, until p independent tasks are created. The recursion tree is computed in breadth-first order. Block multiplication in (5.8) is performed in parallel by all processors available at the current level of recursion. When blocks become sufficiently small, block closure is computed sequentially by a processor chosen arbitrarily from the available processors. The algorithm is as follows.

Algorithm 19. *Algebraic path computation in an acyclic graph.*

Parameter: integer $n \geq p^{2/3}$.

Input: $n \times n$ lower triangular matrix A over a semiring.

Output: $n \times n$ matrix closure A^* (assuming it exists), overwriting A.

Description. The computation is performed on a CRCW BSPRAM(p, g, l), and is defined by recursion on the size of the matrix. Denote the matrix size at the current level of recursion by m, keeping n for the original size. Let $n_0 = n/p^{1/3}$. Value n_0 is the threshold, at which the algorithm switches from parallel to sequential computation.

In each step of recursion, the matrix is divided into regular square blocks of size $m/2$ as shown in (5.7). Then, computation (5.8) is performed by the following schedule.

Small blocks. If $1 \leq m \leq n_0$, choose an arbitrary processor from the currently available, and compute (5.8) on this processor.

Large blocks. If $n_0 < m \leq n$, partition the set of currently available processors into two equal subsets. Compute $\bar{A}_{11}$ (respectively, $\bar{A}_{22}$) by recursion, using the processors of the first (respectively, the second) subset. Then compute $\bar{A}_{21}$ by Algorithm 10, using all processors available at the current level of recursion. Overwrite every A_{ij} by $\bar{A}_{ij}$, obtaining the matrix closure (5.9).

Cost analysis. The values for $W = W_p(n)$, $H = H_p(n)$, $S = S_p(n)$ can be found from the following recurrence relations:

	$n_0 < m \leq n$	$m = n_0$
$W_q(m) =$	$W_{q/2}(m/2) + O(m^3/q)$	$O(n_0^3)$
$H_q(m) =$	$H_{q/2}(m/2) + O\big(m^2/q^{2/3}\big)$	$O(n_0^2)$
$S_q(m) =$	$S_{q/2}(m/2) + O(1)$	$O(1)$

as

$$W = O(n^3/p) \qquad H = O\big(n^2/p^{2/3}\big) \qquad S = O(\log p)$$

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

From the analysis of Algorithm 19, the algebraic path problem on an acyclic graph appears to be asymptotically easier than on a general graph. The asymptotic communication and local computation costs of Algorithm 19 are the same as for matrix multiplication, and the synchronisation cost is higher than that of matrix multiplication by only a factor of $\log p$. There is no communication-synchronisation tradeoff. The proof of a lower bound on communication cost $H = \Omega(n^2/p^{2/3})$ is identical to the proof for general graphs, given in Section 5.2.

If A is a matrix over a commutative ring with unit, fast matrix multiplication can be used instead of standard matrix multiplication for computing block products. The modified algorithm is as follows.

Algorithm 20. *Fast algebraic path computation in an acyclic graph.*

Parameter: integer $n \geq p^{2/\omega}$.

Input: $n \times n$ lower triangular matrix A over a commutative ring with unit.

Output: $n \times n$ matrix closure A^* (assuming it exists), overwriting A.

Description. The computation is identical to Algorithm 19, except that block multiplication is performed by Algorithm 11, rather than Algorithm 10.

Cost analysis. The values for $W = W_p(n)$, $H = H_p(n)$, $S = S_p(n)$ can be found from the following recurrence relations:

	$n_0 < m \leq n$	$m = n_0$
$W_q(m) =$	$W_{q/2}(m/2) + O(m^\omega/q)$	$O(n_0^\omega)$
$H_q(m) =$	$H_{q/2}(m/2) + O\big(m^2/q^{2\omega^{-1}}\big)$	$O(n_0^2)$
$S_q(m) =$	$S_{q/2}(m/2) + O(1)$	$O(1)$

as

$$W = O(n^\omega/p) \qquad H = O\big(n^2/p^{2\omega^{-1}}\big) \qquad S = O(\log p)$$

The algorithm is oblivious, with slackness and granularity $\sigma = \gamma = n^2/p^{2\omega^{-1}}$. ■

If A is a real nonsingular lower triangular matrix, then the algorithm for computing the inverse A^{-1} can be obtained by transforming equations (5.8) into the form

$$\begin{aligned} \bar{A}_{11} &\leftarrow A_{11}^{-1} \\ \bar{A}_{22} &\leftarrow A_{22}^{-1} \\ \bar{A}_{21} &\leftarrow -\bar{A}_{22} A_{21} \bar{A}_{11} \end{aligned} \tag{5.10}$$

after which every $\bar{A}_{ij}$ overwrites A_{ij}. Matrices A_{11}, A_{22} are nonsingular lower triangular, therefore the procedure can be be applied recursively to find A_{11}^{-1} and A_{22}^{-1}. The resulting matrix is

$$A^{-1} = \begin{pmatrix} A_{11}^{-1} & \\ -A_{22}^{-1} A_{21} A_{11}^{-1} & A_{22}^{-1} \end{pmatrix} \tag{5.11}$$

Thus, triangular matrix inversion can be performed by an algorithm similar to Algorithms 19 or 20.

5.4 All-pairs shortest paths computation

In Section 5.2 we considered the algebraic path problem over an arbitrary semiring. Here we deal with a special case where the semiring is the set of real numbers with ∞, and the numerical operations min and + are used as semiring addition and multiplication respectively. Since the min operation is idempotent, for all i, j there is a path from i to j of length $A^*[i,j]$ — this is one of the *shortest paths* from i to j. Most algorithms for matrix closure in the $(\min, +)$ semiring can be extended to compute the shortest paths between all pairs of nodes, as well as the distances. Therefore, in this section we use the term *all-pairs shortest paths problem* as a synonym for the matrix closure problem in the $(\min, +)$ semiring. Symbols + and $\cdot$, when applied to path lengths, will always denote the semiring addition

and multiplication, i.e. numerical min and +. Initially, we consider the case where all edge lengths are nonnegative. We then extend our method to general lengths.

The technique of Gauss–Jordan elimination, considered in Section 5.2, can be applied to the all-pairs shortest paths problem. In this context, Gauss–Jordan elimination is commonly known as the *Floyd–Warshall algorithm* (see e.g. [CLR90]). Its block recursive version, identical to Algorithm 17, solves the problem with BSP cost $W = O(n^3/p)$, $H = O(n^2/p^\alpha)$, $S = O(p^\alpha)$, for an arbitrary α, $1/2 \leq \alpha \leq 2/3$.

Alternatively, the problem with nonnegative lengths can be solved by *Dijkstra's algorithm* ([Dij59], see also [CLR90]). This greedy algorithm finds all shortest paths from a fixed source in order of increasing length. The sequential time complexity of Dijkstra's algorithm is $\Theta(n^2)$. To compute the shortest paths between all pairs of nodes in parallel, one can apply Dijkstra's algorithm independently to each node as a source (this approach is suggested e.g. in [Joh77, Fos95]). The resulting algorithm has BSP cost $W = O(n^3/p)$, $H = O(n^2)$, $S = O(1)$. It thus has a higher communication cost, but a lower synchronisation cost, than the Floyd–Warshall algorithm. This tradeoff motivates us to look for an improved BSP algorithm, that would solve the all-pairs shortest paths problem efficiently both in communication and synchronisation.

In order to design such an algorithm, we use the principle of *path doubling*. No shortest path may contain more than n edges, therefore $A^n = A^*$. Matrix A^n can be obtained by repeated squaring in $\log n$ matrix multiplications. Therefore, the local computation cost of computing A^n by repeated squaring is $W = \Theta\big((n^3 \log n)/p\big)$. A refined version of path doubling was proposed in [AGM97, Tak98]. When run in parallel, this method allows one to compute the matrix $A^n = A^*$ with local computation cost $W = O(n^3/p)$. Compared to the Floyd–Warshall algorithm, the new method does not improve on the synchronisation cost by itself; however, an improvement can be achieved by combining the new method with Dijkstra's algorithm.

We assume for simplicity that all edges and paths in the graph have different lengths, therefore all shortest paths are unique. We use the term *path size* for the number of edges in a path. The main idea of the method is to perform path doubling, keeping track not only of path lengths, but also of path sizes. We assume that lengths and sizes are kept in a single data structure, called the *path matrix*. In such a matrix X, each entry $X[i,j]$ is either ∞, or corresponds to a simple path from i to j. Addition and multiplication of path matrices are defined in the natural way. For an integer k, let $X(k)$ denote the matrix of all paths in X of size exactly k. More precisely, $X(k)[i,j] = X[i,j]$ if path $X[i,j]$ has size k, and $X(k)[i,j] = \infty$ otherwise. Let $X(k_1, \ldots, k_s) = X(k_1) + \cdots + X(k_s)$ (remembering that + denotes numerical min). Note that $X(0) = I$ and $X = X(0, \ldots, m) = I + X(1) + \cdots + X(m)$, where m is the maximum path size in X.

For path matrices X, Y, we write $X \leq Y$, if $X[i,j] \leq Y[i,j]$ for all i,j (ignoring path sizes). We call an entry $X[i,j]$ *trivial*, if $i = j$, or $X[i,j] = \infty$. We call X and Y *disjoint*, if either $X[i,j]$, or $Y[i,j]$ is trivial for all i,j.

For an integer k, matrix A^k contains all shortest paths of size at most k (and maybe some other paths). Suppose that we have computed A^k for some k, $1 \leq k < n$. Our next goal is to compute all shortest paths of size at most $3k/2$. Decompose the path matrix A^k into a disjoint semiring sum $A^k = I + A^k(1) + \cdots + A^k(k)$. Consider the matrices $A^k(k/2+1), \ldots, A^k(k)$. The total number of nontrivial entries in all these matrices is at most n^2 (since the matrices are disjoint), and the average number of nontrivial entries per matrix is at most $2n^2/k$. Therefore, for some l, $k/2 < l \leq k$, matrix $A^k(l)$ contains at most $2n^2/k$ nontrivial entries.

Consider any shortest path of size in the range $l+1, \ldots, 3k/2$. This path consists of an initial subpath of size l, and a final subpath of size at most k. Therefore, the matrix product $A^k(0,l) \cdot A^k$ contains all shortest paths of size at most $3k/2$: $A^k(0,l) \cdot A^k \leq A^{3k/2}$. Since $A^k(0,l)$ has at most $2n^2/k$ nontrivial entries, computation of $A^k(0,l) \cdot A^k$ requires not more than $2n^3/k$ semiring multiplications.

The above product involves a sparse matrix $A^k(0,l)$, and a dense matrix A^k. To compute $A^k(0,l) \cdot A^k$ efficiently in parallel, we need to partition the problem into p sparse-by-dense matrix multiplication subproblems, where all the sparse arguments have an approximately equal number of nontrivial entries. This can be done by first partitioning the set of rows in $A^k(0,l)$ into $p^{1/3}/k^{1/3}$ equal subsets, such that each subset contains at most $\frac{2n^2}{k^{2/3} \cdot p^{1/3}}$ nontrivial entries. This partitioning defines, up to a permutation of rows, a decomposition of the matrix into $p^{1/3}/k^{1/3}$ equal horizontal strips. Each strip defines an $\frac{n \cdot k^{1/3}}{p^{1/3}} \times n \times n$ sparse-by-dense matrix multiplication subproblem.

Consider one of the above subproblems. Partition the set of columns in the strip into $p^{1/3}/k^{1/3}$ equal subsets, such that each subset contains at most $\frac{4n^2}{k^{1/3} \cdot p^{2/3}}$ nontrivial entries. This partitioning defines, up to a permutation of columns, a decomposition of the strip into equal square blocks. Each block defines an $\frac{n \cdot k^{1/3}}{p^{1/3}} \times \frac{n \cdot k^{1/3}}{p^{1/3}} \times n$ sparse-by-dense matrix multiplication subproblem. By partitioning the set of columns of the second argument of this subproblem into $p^{1/3} \cdot k^{2/3}$ equal subsets, we obtain $p^{1/3} \cdot k^{2/3}$ sparse-by-dense matrix multiplication subproblems of size $\frac{n \cdot k^{1/3}}{p^{1/3}} \times \frac{n \cdot k^{1/3}}{p^{1/3}} \times \frac{n}{p^{1/3} \cdot k^{2/3}}$.

The total number of resulting sparse-by-dense matrix multiplication subproblems is p. The sparse argument of each subproblem contains at most $\frac{4n^2}{k^{1/3} \cdot p^{2/3}}$ nontrivial entries. The partitioning can be computed by a greedy algorithm, the BSP cost of which is negligible. The BSP cost of computing the matrix product $A^k(0,l) \cdot A^k$ is therefore $W = O\big(n^3/(k \cdot p)\big)$, $H = O\big(n^2/(k^{1/3} \cdot p^{2/3})\big)$, $S = O(1)$.

The path doubling process stops after at most $\log_{3/2} p$ rounds, when the matrix A^p (or some matrix $\leq A^p$, which is only better) has been computed. For some q, $1 \leq q \leq p$, matrix $A^p(q)$ contains at most n^2/p nontrivial entries. Therefore, it can be broadcast to every processor with communication cost $H = O(n^2/p)$. Each processor receives the matrix $A^p(q)$, picks n/p nodes, and computes all shortests paths originating in these nodes by n/p independent runs of Dijkstra's algorithm. The result of this computation across all processors is the matrix closure $A^p(q)^*$. Matrix $A^p(q)^*$ contains all shortest paths of sizes that are multiples of q (and maybe some other paths).

Any shortest path in A^* consists of an initial subpath of size that is a multiple of q, and a final subpath of size at most $q \leq p$. Therefore, all shortest paths for the original matrix A can be computed as the matrix product $A^p(q)^* \cdot A^p = A^*$.

The cost of the resulting algorithm is $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, $S = O(\log p)$. We can further reduce the synchronisation cost by terminating the path doubling phase after fewer than $\log_{3/2} p$ steps. For $1 \leq r \leq p^{2/3}$, we can find a q such that the matrix $A^r(q)$ has at most n^2/r nontrivial entries, therefore the communication cost of applying Dijkstra's algorithm to find $A^r(q)^*$ is $H = O(n^2/r)$.

The resulting algorithm is as follows.

Algorithm 21. *All-pairs shortest paths (nonnegative case).*

Parameters: integer $n \geq p$; integer r, $1 \leq r \leq p^{2/3}$.

Input: $n \times n$ matrix A over the $(\min, +)$ semiring of nonnegative real numbers with ∞.

Output: $n \times n$ matrix closure A^*.

Description. The computation is performed on a CRCW BSPRAM(p, g, l), and proceeds in three stages.

First stage. Compute A^r and $A^r(q)$, $0 < q \leq r$, by at most $\log_{3/2} r$ rounds of path doubling. Matrix $A^r(q)$ contains at most n^2/r nontrivial entries.

Second stage. Broadcast $A^r(q)$ and compute the closure $A^r(q)^*$ by n independent runs of Dijkstra's algorithm, n/p runs per processor.

Third stage. Compute the product $A^r(q)^* \cdot A^r = A^*$.

Cost analysis. The local computation and communication costs of the first stage are dominated by the cost of its first round: $W = O(n^3/p)$ and $H = O(n^2/p^{2/3})$. The synchronisation cost of the first stage is $S = O(\log r)$.

The cost of the second stage is $W = O(n^3/p)$, $H = O(n^2/r)$, $S = O(1)$. The cost of the third stage is $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, $S = O(1)$.

The local computation, communication and synchronisation costs of the whole algorithm are

$$W = O(n^3/p) \qquad H = O(n^2/r) \qquad S = O(1 + \log r)$$

The algorithm is communication-oblivious, with slackness $\sigma = n^2/r$ and granularity $\gamma = 1$. ■

The two extremes of Algorithm 21 are the communication-efficient algorithm with $r = p^{2/3}$, $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, $S = O(\log p)$, and the multiple Dijkstra algorithm with $r = 1$, $W = O(n^3/p)$, $H = O(n^2)$, $S = O(1)$.

Algorithm 21 allows the following variation. Instead of computing the closure $A^p(q)^*$, represent matrix $A^p(p)$ as a product $A^p(p) = A^p(q) \cdot A^p(p-q)$, $0 \leq q < p/2$. For some q, the disjoint sum $A^p(q) + A^p(p-q)$ contains at most $2n^2/p$ nontrivial entries. Therefore, the second stage of the algorithm can be replaced by broadcasting the matrices $A^p(q)$ and $A^p(p-q)$ (or, equivalently, their disjoint sum), recovering the product $A^p(q) \cdot A^p(p-q) = A^p(p)$, and computing the closure $A^p(p)^*$. In the third stage, it remains to compute the product $A^p(p)^* \cdot A^p = A^*$.

We now extend Algorithm 21 to graphs where edge lengths may be negative. Formally, the problem is defined as finding the closure A^* of a matrix A over the $(\min, +)$ semiring of all real numbers with ∞. The closure is defined, if and only if the graph does not contain a cycle of negative length. We cannot use our original method to solve this more general problem, because Dijkstra's algorithm does not work on graphs with negative edge lengths. However, we can get round this difficulty by replacing Dijkstra's algorithm with an extra stage of sequential path doubling.

The extended algorithm has three stages. In the first stage, we compute the matrix A^{p^2} by $2\log_{3/2} p$ steps of parallel path doubling. Let $A^{p^2}((p)) = A^{p^2}(p, 2p, \ldots, p^2)$, and $A^{p^2}((p) - q) = A^{p^2}(p - q, 2p - q, \ldots, p^2 - q)$. We represent matrix $A^{p^2}((p))$ as a product $A^{p^2}((p)) = A^{p^2}(q) \cdot A^{p^2}((p) - q)$, $0 \leq q < p/2$. For some q, the disjoint sum $A^{p^2}(q) + A^{p^2}((p) - q)$ contains at most $2n^2/p$ nontrivial entries. In the second stage, we collect matrices $A^{p^2}(q)$ and $A^{p^2}((p) - q)$ in a single processor, and recover their product $A^{p^2}((p))$. Now the closure $A^{p^2}((p))^* = A^{p^2}(p)^*$ can be computed by sequential path doubling. In the third stage, it remains to compute the product $A^{p^2}(p)^* \cdot A^{p^2} = A^*$.

In contrast with the nonnegative case, early termination of the parallel path doubling phase would increase not only the communication cost, but also the local computation cost. Therefore, we do not consider this option.

The resulting algorithm is as follows.

Algorithm 22. *All-pairs shortest paths.*

Parameter: integer $n \geq p$.

Input: $n \times n$ matrix A over the $(\min, +)$ semiring of real numbers with ∞.

Output: $n \times n$ matrix closure A^*.

Description. The computation is performed on a CRCW BSPRAM(p, g, l), and proceeds in three stages.

First stage. Compute A^{p^2} and $A^{p^2}((p))$ by at most $2\log_{3/2} p$ rounds of parallel path doubling. Represent $A^{p^2}((p))$ as $A^{p^2}((p)) = A^{p^2}(q) \cdot A^{p^2}((p) - q)$, $0 \leq q \leq p/2$. The disjoint sum $A^{p^2}(q) + A^{p^2}((p) - q)$ contains at most $2n^2/p$ nontrivial entries.

Second stage. Collect $A^{p^2}(q) + A^{p^2}((p) - q)$ in a single processor, and recover $A^{p^2}((p)) = A^{p^2}(q) \cdot A^{p^2}((p) - q)$. Compute the closure $A^{p^2}((p))^* = A^{p^2}(p)^*$ by sequential path doubling.

Third stage. Compute the product $A^{p^2}(p)^* \cdot A^{p^2} = A^*$.

Cost analysis. The local computation and communication costs of the first stage are dominated by the cost of its first round: $W = O(n^3/p)$ and $H = O(n^2/p^{2/3})$. The synchronisation cost of the first stage is $S = O(\log p)$.

The local computation cost of the second stage is dominated by the cost of its first round, equal to $W = O(n^3/p)$. The communication and synchronisation costs of the second stage are $H = O(n^2/p)$, $S = O(1)$.

The cost of the third stage is $W = O(n^3/p)$, $H = O(n^2/p^{2/3})$, $S = O(1)$.

The local computation, communication and synchronisation costs of the whole algorithm are

$$W = O(n^3/p) \qquad H = O(n^2/p^{2/3}) \qquad S = O(\log p)$$

The algorithm is communication-oblivious, with slackness $\sigma = n^2/p$ and granularity $\gamma = 1$. ■

The described method is applicable not only to the $(\min, +)$ semiring, but also to any semiring where addition is idempotent. The examples are the $(\vee, \wedge)$ semiring for the problem of transitive closure, the $(\max, \min)$ semiring for paths of maximum capacity, or the $(\max, \cdot)$ semiring for paths of maximum reliability. Note that in the case of transitive closure computation by Algorithm 21, Boolean matrix multiplication (Algorithm 16) cannot be used instead of general matrix multiplication (Algorithm 10), since the path doubling process involves the multiplication of path matrices, rather than ordinary Boolean matrices. It is not clear if an extension of Algorithm 16 can be obtained for path matrix multiplication.

5.5 Single-source shortest paths computation

In the previous section, we presented deterministic BSP algorithms for finding all-pairs shortest paths in a dense weighted graph. In this section, we

deal with the problem of finding all shortest paths from a single distinguished node, called *the source*. The considered graph may be sparse, and some edge lengths may be negative.

We develop an efficient BSP algorithm for the single-source shortest paths problem, by using a simple randomisation method proposed in [UY91]. First, we choose a random subset of $s \leq n$ *sample nodes*, which also includes the source node. If $s \geq n^\epsilon$ for a constant $\epsilon > 0$, then an arbitrary subset of $n \log n/s$ nodes contains a sample node with high probability (see e.g. [UY91]). For every sample node, we then compute all outgoing shortest paths of size at most $n \log n/s$. Consider any shortest path beginning at the source. With high probability, this path is divided by the sample nodes into shortest subpaths of size at most $n \log n/s$. These subpaths are among the shortest paths just computed. Let the *sample graph* be defined as the graph on the sample nodes, with edges corresponding to the shortest paths between the samples. The next step of the algorithm is to compute all-pairs shortest paths in the sample graph. After that, one matrix multiplication is sufficient to complete the computation of single-source shortest paths in the original graph.

We assume that the input is a sparse graph with m edges, $n \leq m \leq n^2$. The high-probability argument requires that $m = \Omega(n^{1+\epsilon})$ for some constant $\epsilon > 0$. As before, the graph is weighted and directed, with arbitrary real edge lengths. We keep using the notation $+$ and $\cdot$, when applied to edge lengths, for numerical min and $+$ respectively.

The single-source shortest path problem in a graph corresponds to a system of linear equations in the (min, +) semiring. The standard Bellman–Ford single-source shortest paths algorithm (see e.g. [CLR90]) can be viewed as solving this linear system by Jacobi iteration. An iteration step has sequential cost $O(m)$, and consists in multiplying the system matrix by a vector. A total of n steps may be required, therefore the sequential cost of the Bellman–Ford algorithm is $O(mn)$.

The randomised shortest paths algorithm needs to compute shortest paths from s different sources, up to path size $n \log n/s$. This corresponds to $n \log n/s$ steps of Jacobi iteration, performed independently on s initial vectors. Alternatively, the computation can be viewed as a sequence of $n \log n/s$ steps, each of which is a multiplication of a sparse $n \times n$ matrix by a dense $n \times s$ matrix. The sequential cost of this computation is $O(mn \log n)$.

Let A be the graph matrix. Without loss of generality, we assume that the source node has index 1. Single-source shortest paths can therefore be computed by the following simple, synchronisation-efficient randomised BSP algorithm.

Algorithm 23. *Randomised single-source shortest paths in sparse graph.*

Parameter: integer $n \geq p^2$.

Input: $n \times n$ matrix A over the $(\min, +)$ semiring of real numbers with ∞. Matrix A contains $m = \Omega(n^{1+\epsilon})$ nontrivial entries, for a constant $\epsilon > 0$.

Output: n-vector $(0, \infty, \ldots, \infty) \cdot A^*$.

Description. Let $s = \min(m/n, n^{1/2})$. The computation is performed on a CRCW BSPRAM(p, g, l), and proceeds in three stages.

First stage. Broadcast matrix A. Select s random sample nodes, including the source node. Without loss of generality, let the sample nodes have indices $1, \ldots, s$. Let J denote an $n \times s$ matrix, such that $J[i, j] = 0$ if $i = j$, and $J[i, j] = \infty$ otherwise. Compute $s \times n$ matrix $B = J^T \cdot A^{n \log n/s}$ by s independent runs of the Bellman–Ford algorithm. Compute $s \times s$ matrix $C = B \cdot J = J^T \cdot A^{n \log n/s} \cdot J$.

Second stage. Collect matrix C in a single processor, and compute its closure C^* by an efficient sequential algorithm. Let the s-vector c be the first row of C^*: $c = (0, \infty, \ldots, \infty) \cdot C^*$.

Third stage. Broadcast c, and compute the n-vector $c \cdot B$. With high probability, this vector is equal to the first row of A^*: $c \cdot B = (0, \infty, \ldots, \infty) \cdot A^*$.

Cost analysis. The communication cost of broadcast in the first stage is $H = O(m)$. The local computation cost of the first stage is determined by the cost of the Bellman–Ford algorithm: $W = O(mn \log n/p)$.

The communication cost of collecting the sample matrix C in the second stage is $H = O(s^2)$. The local computation cost of finding the closure C^* is $W = O(s^3)$. Both costs are dominated by the respective costs of the first stage.

The communication cost of broadcast in the third stage is $H = O(s)$. The local computation cost of the third stage is $W = O(sn/p)$. Again, both costs are dominated by the respective costs of the first stage.

The local computation, communication and synchronisation costs of the whole algorithm are

$$W = O(mn \log n/p) \qquad H = O(m) \qquad S = O(1)$$

The algorithm is communication-oblivious, with slackness and granularity $\sigma = \gamma = s/p$. ■

The local computation cost of Algorithm 23 differs from the cost of the sequential Bellman–Ford algorithm by a factor of $\log n$. It is not known whether the single-source shortest path problem can be solved in parallel with local computation cost $O(mn/p)$.

The communication cost of Algorithm 23 can be reduced by partitioning matrix A in the first stage into regular square blocks, and performing the computation of matrix B in $n \log n/s$ supersteps. We do not consider this alternative fine-grain algorithm, because of its high synchronisation cost.

5.6 Minimum spanning tree computation

Finally, we consider the problem of finding the minimum spanning tree (MST) in a weighted undirected graph. As observed in [MP88], the MST problem can be viewed as an instance of the algebraic path problem. The problem input is a symmetric matrix over the semiring of all real numbers with ∞, where the operations min and max are used as semiring addition and multiplication respectively. Due to the symmetry of the matrix, and the special structure of the $(\min, \max)$ semiring, the MST can be found in sequential time $O(n^2)$, in contrast with the $\Theta(n^3)$ complexity of standard algorithms for the general algebraic path problem.

Standard sequential algorithms for MST computation employ greedy techniques, such as the algorithms by Kruskal and Prim (see e.g. [Chr75, AHU83, CLR90]). These greedy algorithms find the MST of a graph with n nodes and m edges in time $O(m \log n)$. Many algorithms with a lower asymptotic complexity have been proposed, but it is not known if an $O(m)$ deterministic algorithm exists. However, if the input edges are sorted by weight, the greedy algorithms work in time $O(m)$. Paper [KKT95] describes a randomised $O(m)$ MST algorithm.

A standard PRAM solution to the problem is provided by another greedy algorithm, attributed to Borůvka and Sollin (see e.g. [JáJ92]). The algorithm works by selecting for each node the shortest incident edge. The resulting set of edges is a subforest of the MST. Connected components of this forest are regarded as nodes of a new graph, where the weight of an edge between two nodes is defined as the minimum weight of an edge between the two corresponding components. The procedure is repeated until only one node is left. The MST of the original graph is the union of the forests constructed in all rounds. The number of nodes in the graph is reduced by at least a factor of two in each round, therefore $O(\log n)$ rounds are sufficient for a dense graph. The contraction of tree components in each round can take up to $O(\log n)$ steps, therefore the total PRAM complexity of the algorithm is $O(\log^2 n)$. Paper [JM95] presents a more efficient PRAM algorithm, with complexity $O(\log^{3/2} n)$.

The BSPRAM model suggests an alternative, coarse-grain approach to the problem. We assume that initially the edges of the graph are stored in the main memory. We also assume that all edge weights are distinct (otherwise we can break the ties by attaching a unique identifier to each edge). A useful theorem from [MP88] relates the MST problem to the problem of finding shortest paths in a graph.

Theorem 11 (Maggs, Plotkin). *A path between two nodes in the minimum spanning tree is the lightest path between these nodes, where path weight is defined as the weight of the heaviest edge in the path.*

Proof. See [MP88]. ■

To find the MST, consider a partitioning of the m edges into p arbitrary subsets. Each subset defines a subgraph of the original graph. We compute the MST of each subgraph separately by an efficient sequential algorithm, either deterministic or randomised. Any edge that does not belong to one of the subgraph MSTs does not belong to the MST of the whole graph. Indeed, such an edge cannot be the lightest path between its ends, since there is a lighter path in the MST of the subgraph that contains that edge. Therefore, the MST of the whole graph is contained in the union of subgraph MSTs, and can be found as the MST of this union. The resulting BSPRAM algorithm is as follows.

Algorithm 24. *Minimum spanning tree.*

Parameters: integer $n \geq p^2$ (respectively, $n \geq p^2 \log p$); integer $m \geq n \cdot p^2$ (respectively, $m \geq n \cdot p^2 \log p$, $m \geq n \log n$) for the randomised (respectively, deterministic) version of the algorithm.

Input: undirected weighted graph G with n nodes and m edges.

Output: minimum spanning tree of G.

Description. The computation is performed on an EREW BSPRAM(p, g, l). Edges of G are partitioned across the processors, m/p edges per processor. After that, the computation is performed in message-passing mode and proceeds in two stages.

First stage. Each processor computes the MST of the subgraph of G defined by the m/p local edges, and then sorts the edges of the obtained local MST.

Second stage. The local MSTs are collected in a single processor. This processor merges the received edges into a sorted sequence, and then computes the MST of their union, obtaining the MST of G.

Cost analysis. The communication cost of the initial distribution is $H = O(m/p)$. The (expected) local computation cost of computing subgraph MSTs in the first stage is $O(m \log n/p)$ for the deterministic version, and $O(m/p)$ for the randomised version. The cost of sorting the edges of the local MSTs is $O(n \log n/p)$. The size of each local MST is $O(n)$. Therefore, the communication cost of collecting the local MSTs is $O(n \cdot p)$. The local computation cost of merging the local MSTs in the second stage is $O(n \cdot p \log p)$. The assumptions on the size of n and m imply that the BSP cost of the second stage is dominated by the cost of the first stage. Thus, the local computation cost of the deterministic algorithm is $W_{\text{det}} = O(m \log n/p)$, and the expected local computation cost of the randomised algorithm is $W_{\text{exp}} = O(m/p)$. The communication and synchronisation costs of both algorithms are $H = O(m/p)$, $S = O(1)$. The algorithm is communication-oblivious. Its slackness and granularity are $\sigma = \gamma = m/p$. ■

Papers [ADJ+98, DG98] present more complex BSP algorithms, that are efficient for smaller input sizes. For dense graphs, these algorithms are similar to Algorithm 24.

Chapter 6

Conclusions

In this thesis we have developed a systematic approach to the design and analysis of bulk-synchronous parallel algorithms, based on the BSPRAM model. This model enhances the standard BSP model with shared memory, while retaining the concept of data locality. It was shown that the BSP and the BSPRAM models are related by efficient simulation for a broad range of algorithms. We have identified some characteristic algorithm properties that enable such simulation: communication-obliviousness, slackness, granularity. The BSPRAM approach encourages natural specification of the problems: the input and output data are assumed to be stored in the main memory, no assumptions on data distribution are necessary. The use of shared memory simplifies the design and analysis of BSP algorithms.

We have presented BSPRAM algorithms for popular computational problems from three large domains: combinatorial computation, computation on dense matrices, and graph computation. The BSP costs of the presented algorithms are summarised in Tables 6.1–6.4 (with constant factors omitted). It is assumed that the input size is sufficiently large with respect to the number of processors. Some of the algorithms exhibit a tradeoff between communication and synchronisation costs. Constant factors involved in the asymptotic costs of the presented algorithms are fairly small, with the exception of fast Boolean matrix multiplication, where these factors are astro-

Problem	W	H	S
Complete tree	n/p	n/p	1
Butterfly dag	$n \log n/p$	n/p	1
Cube dag	n^3/p	$n^2/p^{1/2}$	$p^{1/2}$
Sorting	$n \log n/p$	n/p	1
List contraction	n/p	n/p	$\log p$
Tree contraction	n/p	n/p	$\log p$

Table 6.1: Summary of combinatorial algorithms

Problem	W	H	S
Matrix-vector multiplication	n^2/p	$n/p^{1/2}$	1
Triangular system solution	n^2/p	n	p
Matrix multiplication	n^3/p	$n^2/p^{2/3}$	1
Gaussian elimination	n^3/p		
no pivoting, min H	–	$n^2/p^{2/3}$	$p^{2/3}$
no pivoting, min S	–	$n^2/p^{1/2}$	$p^{1/2}$
block pivoting, min H	–	$n^2/p^{2/3}$	$p^{2/3} \cdot (\log p)^2$
block pivoting, min S	–	$n^2 \cdot \frac{(\log p)^{1/2}}{p^{1/2}}$	$p^{1/2} \cdot (\log p)^{3/2}$
column pivoting	–	n^2	p

Table 6.2: Summary of matrix algorithms

Problem	W	H	S
Matrix multiplication	n^ω/p	$n^2/p^{2/\omega}$	1
Gaussian elimination	n^ω/p		
no pivoting, min H	–	$n^2/p^{2/\omega}$	$p^{2/\omega}$
no pivoting, min S	–	$n^2/p^{1/2}$	$p^{1/2}$
Boolean matrix mult	n^ω/p		
normal	–	$n^2/p^{2/\omega}$	1
min H, any W	$n^\omega \cdot \exp(p^{68})$	n^2/p	1

Table 6.3: Summary of fast matrix algorithms

Problem	W	H	S
Algebraic paths	n^3/p		
general, min H	–	$n^2/p^{2/3}$	$p^{2/3}$
general, min S	–	$n^2/p^{1/2}$	$p^{1/2}$
acyclic	–	$n^2/p^{2/3}$	$\log p$
All-pairs shortest paths	n^3/p		
general	–	$n^2/p^{2/3}$	$\log p$
nonnegative, min H	–	$n^2/p^{2/3}$	$\log p$
nonnegative, min S	–	n^2	1
Single-source shortest paths	$mn \log n/p$		
general, randomised	–	m	1
Minimum spanning tree	m/p		
randomised	–	m/p	1
deterministic	$m \log n/p$	m/p	1

Table 6.4: Summary of graph algorithms

nomically large. The question remains whether a practical communication-efficient algorithm for Boolean matrix multiplication exists.

Asymptotic optimality has been proven for some of the presented algorithms. For other algorithms, we have discussed possible methods of obtaining a lower bound. Lower bounds on communication and synchronisation were treated separately. A possible direction of future research consists in obtaining lower bounds on communication-synchronisation tradeoffs. Other research directions include a numerical stability study for the dense matrix algorithms obtained in Chapter 4, and an extension of these algorithms to sparse matrices.

Our presentation highlights certain algorithmic concepts that are important in developing future BSP languages and programming tools. Apart from communication-synchronisation tradeoffs, these concepts include blocking and recursion. Providing their efficient support both on the local and the inter-processor level is a challenging task for future BSP software development. Implementation of the BSPRAM model and experimental evaluation of the algorithms presented in this thesis would be useful first steps in this direction.

Chapter 7

Acknowledgements

This work began in 1993, when I was a visiting student at Oxford University Computing Laboratory, funded by the Soros/FCO Scholarship. My subsequent DPhil studies were supported in part by ESPRIT Basic Research Project 9072 GEPPCOM — Foundations of General Purpose Parallel Computing. I am grateful to my supervisor Bill McColl for the exciting opportunity to participate in this project, and for many illuminating discussions. I thank my examiners Richard Brent and Mike Paterson for their attentive reading and valuable advice. Further thanks are due to Rob Bisseling, Radu Calinescu, Alex Gerbessiotis, Ben Juurlink, Jimmie Lawson, Constantinos Siniolakis, Karina Terekhova, Alex Wagner, as well as the anonymous referees of my several papers. The thesis was typeset in LaTeX 2_ε with packages AMS-LaTeX and PSTricks. Thanks to my Oxford colleagues for the stimulating atmosphere, to my friends for making my life in Oxford full and enjoyable, to my wife Karina and to all my family for their love, support and understanding.

Appendix A

The Loomis–Whitney inequality and its generalisations

The optimality of several BSP matrix algorithms presented in Chapter 4 rests on the discrete Loomis–Whitney inequality. The latter is closely related to Cauchy's inequality and the classical isoperimetric inequality. In this appendix we propose a new simple proof and a generalisation of the discrete Loomis–Whitney inequality.

One of the most common inequalities in analysis is the classical *Cauchy inequality* (see e.g. [Mit70]): for two finite sequences $a_i, b_i \geq 0$, $1 \leq i \leq n$,

$$\sum_{i=1}^{n} a_i^{1/2} b_i^{1/2} \leq \left(\sum_{i=1}^{n} a_i \right)^{1/2} \left(\sum_{i=1}^{n} b_i \right)^{1/2} \tag{A.1}$$

A standard generalisation of (A.1) for exponents α and β, $\alpha + \beta = 1$, is Hölder's inequality. It can be further generalised by taking $m \geq 2$ sequences, and exponents $\alpha_1, \ldots, \alpha_m$, such that $\alpha_1 + \cdots + \alpha_m = 1$.

Another important result is the *Loomis–Whitney inequality*, relating the volume of a compact set in $\mathbf{R}^m$, $m \geq 2$, to the areas of its orthogonal projections onto r-dimensional coordinate subspaces, $1 \leq r \leq m$. It was introduced in [LW49] (see also [Had57, BZ88]) to simplify the proof of the classical isoperimetric "volume-to-surface" inequality. The discrete analog of the Loomis–Whitney inequality relates the size of a finite set of points in $\mathbf{Z}^m$ to the sizes of its orthogonal projections onto r-dimensional coordinate subspaces.

For simplicity, let us take $m = 3$, $r = 2$. Let A be a finite set of points in $\mathbf{Z}^3$, and let A_1, A_2, A_3 be the orthogonal projections of A onto the coordinate planes. The discrete Loomis–Whitney inequality states that

$$|A| \leq |A_1|^{1/2} \cdot |A_2|^{1/2} \cdot |A_3|^{1/2} \tag{A.2}$$

where $|\cdot|$ denotes the cardinality of a finite set.

Inequalities (A.1) and (A.2) seem at first to be unrelated. However, they are special cases of the following general inequality: for finite sequences $a_{ij}, b_{jk}, c_{ik} \geq 0$, $1 \leq i, j, k \leq n$,

$$\sum_{i,j,k=1}^{n} a_{ij}^{1/2} b_{jk}^{1/2} c_{ik}^{1/2} \leq \left(\sum_{i,j=1}^{n} a_{ij} \right)^{1/2} \left(\sum_{j,k=1}^{n} b_{jk} \right)^{1/2} \left(\sum_{i,k=1}^{n} c_{ik} \right)^{1/2} \tag{A.3}$$

The Cauchy inequality (A.1) follows from (A.3) when $a_{ij} = a_j$ for all i, $b_{jk} = b_j$ for all k, and $c_{ki} = 1$ for all i, k. The discrete Loomis–Whitney inequality (A.2) follows from (A.3) when $a_{ij}, b_{jk}, c_{ki} \in \{0, 1\}$.

Inequality (A.3) can be rewritten in matrix form as $\mathrm{tr}(ABC) \leq \|A\| \cdot \|B\| \cdot \|C\|$, where $A = \left(a_{ij}^{1/2}\right)$, $B = \left(b_{jk}^{1/2}\right)$, $C = \left(c_{ki}^{1/2}\right)$, and $\|\cdot\|$ is the Frobenius (Euclidean) matrix norm: $\|X\| = \left(\sum_{ij} x_{ij}^2\right)^{1/2}$, where $X = (x_{ij})$. In this form, the inequality can be proved straightforwardly from the properties of the Frobenius norm, and of the Frobenius inner product $\langle X, Y \rangle = \mathrm{tr}(XY^T) \leq \|X\| \cdot \|Y\|$. We have $\mathrm{tr}(ABC) \leq \|A\| \cdot \|(BC)^T\| \leq \|A\| \cdot \|B\| \cdot \|C\|$.

We now consider the multidimensional versions of the above inequalities. Let $m \geq 3$, $1 \leq r \leq m$. For $m-1$ sequences $a_s[i] \geq 0$, $1 \leq s \leq m-1$, $1 \leq i \leq n$, and exponents $\alpha_1 = \cdots = \alpha_{m-1} = \frac{1}{m-1}$, the generalised Hölder inequality is

$$\sum_{i=1}^{n} \prod_{s=1}^{m-1} a_s[i]^{\frac{1}{m-1}} \leq \prod_{s=1}^{m-1} \left(\sum_{i=1}^{n} a_s[i] \right)^{\frac{1}{m-1}}. \tag{A.4}$$

For a finite set $A \subset \mathbf{Z}^m$, the discrete Loomis–Whitney inequality is

$$|A| \leq \textstyle\prod_{1 \leq s_1 < \cdots < s_r \leq m} |A_{s_1 \ldots s_r}|^{mr^{-1}\binom{m}{r}^{-1}}, \tag{A.5}$$

where $A_{s_1 \ldots s_r}$, $1 \leq s_1 < \cdots < s_r \leq m$, are the projections of A onto $\binom{m}{r}$ coordinate subspaces of dimension r.

Inequalities (A.4) and (A.5) are special cases of the following more general inequality.

Theorem 12. *For any* $a_{s_1 \ldots s_r}[i_{s_1}, \ldots, i_{s_r}] \geq 0$, $1 \leq s_1 < \cdots < s_r \leq m$, $1 \leq i_1, \ldots, i_m \leq n$,

$$\textstyle\sum_{1 \leq i_1, \ldots, i_m \leq n} \prod_{1 \leq s_1 < \cdots < s_r \leq m} a_{s_1 \ldots s_r}[i_{s_1}, \ldots, i_{s_r}]^{mr^{-1}\binom{m}{r}^{-1}} \leq$$
$$\textstyle\prod_{1 \leq s_1 < \cdots < s_r \leq m} \left(\sum_{1 \leq i_{s_1}, \ldots, i_{s_r} \leq n} a_{s_1 \ldots s_r}[i_{s_1}, \ldots, i_{s_r}] \right)^{mr^{-1}\binom{m}{r}^{-1}}. \tag{A.6}$$

Inequality (A.3) corresponds to (A.6) with $m = 3$, $r = 2$, $a_{12}[i_1, i_2] = a_{ij}$, $a_{23}[i_2, i_3] = b_{jk}$, $a_{13}[i_1, i_3] = c_{ik}$.

Proof. The matrix proof of inequality (A.3) does not seem to generalise to a proof of (A.6) (although the inequality for $\mathrm{tr}(ABC)$ can be extended to an arbitrary number of matrices). To avoid dealing with the cumbersome notation of (A.6), we give an alternative, elementary proof of (A.3), that can be easily generalised to a proof of (A.6).

We apply Cauchy's inequality (A.1) to the left-hand side of (A.3) three times. The subexpressions to which Cauchy's inequality is applied are denoted below by square brackets. We have

$$\sum_{ijk} a_{ij}^{1/2} b_{jk}^{1/2} c_{ik}^{1/2} =$$

$$\sum_{ik} c_{ik}^{1/2} \left[\sum_{j} a_{ij}^{1/2} b_{jk}^{1/2} \right] \le \sum_{ik} c_{ik}^{1/2} \left(\sum_{j} a_{ij} \right)^{1/2} \left(\sum_{j} b_{jk} \right)^{1/2} =$$

$$\sum_{k} \left(\sum_{j} b_{jk} \right)^{1/2} \left[\sum_{i} c_{ik}^{1/2} \left(\sum_{j} a_{ij} \right)^{1/2} \right] \le$$

$$\sum_{k} \left(\sum_{j} b_{jk} \right)^{1/2} \left(\sum_{i} c_{ik} \right)^{1/2} \left(\sum_{ij} a_{ij} \right)^{1/2} =$$

$$\left(\sum_{ij} a_{ij} \right)^{1/2} \left[\sum_{k} \left(\sum_{j} b_{jk} \right)^{1/2} \left(\sum_{i} c_{ik} \right)^{1/2} \right] \le$$

$$\left(\sum_{ij}^{n} a_{ij} \right)^{1/2} \left(\sum_{jk}^{n} b_{jk} \right)^{1/2} \left(\sum_{ik}^{n} c_{ik} \right)^{1/2}$$

A generalisation for arbitrary r and m is straightforward. ■

Inequality (A.6) degenerates to an identity when $r = 1$ or $r = m$. The generalised symmetric Hölder inequality (A.4) follows from (A.6) when $r = m - 1$,

$$a_{1\ldots k-1,k+1\ldots m-1}[i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_{m-1}] = a_k[i_{m-1}]$$

for $1 \le k < m - 1$, and $a_{1\ldots m-2}[i_1, \ldots, i_{m-2}] = 1$. The discrete Loomis–Whitney inequality (A.5) follows from (A.6) when $a_{s_1\ldots s_r}[i_{s_1}, \ldots, i_{s_r}] \in \{0, 1\}$.

Bibliography

[ABG+95] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–581, 1995.

[ABK95] M. Adler, J. W. Byers, and R. M. Karp. Parallel sorting with limited bandwidth. In *Proceedings of ACM SPAA '95*, pages 129–136, 1995.

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

[ADJ+98] M. Adler, W. Dittrich, B. Juurlink, M. Kutylowski, and I. Rieping. Communication-optimal parallel minimum spanning tree algorithms. In *Proceedings of the 10th ACM SPAA*, 1998.

[ADL+94] N. Alon, R. A. Duke, H. Lefmann, V. Rödl, and R. Yuster. The algorithmic aspects of the regularity lemma. *Journal of Algorithms*, 16(1):80–109, January 1994.

[AGL+98] G. A. Alverson, W. G. Griswold, C. Lin, D. Notkin, and L. Snyder. Abstractions for portable, scalable parallel programming. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):71–86, January 1998.

[AGM97] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences*, 54(2):255–262, April 1997.

[AHU76] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1976.

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley, 1983.

[AM90] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list ranking. *Information Processing Letters*, 33(5):269–273, January 1990.

[BCS97] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*. Number 315 in Grundlehren der mathematischen Wissenschaften. Springer, 1997.

[Ber85] C. Berge. *Graphs*, volume 6, part 1 of *North-Holland Mathematical Library*. North-Holland, second revised edition, 1985.

[BH74] J. R. Bunch and J. E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 21(125):231–236, January 1974.

[BK82] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, 31(3):260–264, March 1982.

[BKM95] J. Basch, S. Khanna, and R. Motwani. On diameter verification and Boolean matrix multiplication. Technical Report STAN-CS-95-1544, Department of Computer Science, Stanford University, 1995.

[Ble93] G. E. Blelloch. Prefix sums and their applications. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 1, pages 35–60. Morgan Kaufmann, 1993.

[BM93] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. Preprint 836, Department of Mathematics, University of Utrecht, December 1993.

[Bol78] B. Bollobás. *Extremal Graph Theory*. Academic Press, 1978.

[Bre91] R. P. Brent. Parallel algorithms in linear algebra. In *Proceedings of the Second NEC Research Symposium*, August 1991.

[BZ88] Yu. D. Burago and V. A. Zalgaller. *Geometric Inequalities*. Number 285 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1988.

[Car79] B. Carré. *Graphs and Networks*. Oxford Applied Mathematics and Computer Science Series. Clarendon Press, 1979.

[CDO+96] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. Clint Whaley. The design and implementation of the ScaLAPACK LU, QR and Cholesky factorization routines. *Scientific Programming*, 5:173–184, 1996.

[Chr75] N. Christofides. *Graph theory: an algorithmic approach*. Computer science and applied mathematics. Academic Press, 1975.

[CLR90] 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.

[Col93] R. Cole. Parallel merge sort. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 10, pages 453–495. Morgan Kaufmann, 1993.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing with application to optimal parallel list ranking. *Information and Control*, 70(1):32–53, July 1986.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, March 1990.

[DG98] F. Dehne and S. Götz. Practical parallel algorithms for minimum spanning trees. In *Proceedings of the Workshop on Advances in Parallel and Distributed Systems*, 1998.

[DHS95] J. W. Demmel, N. J. Higham, and R. S. Schreiber. Block LU factorization. *Numerical Linear Algebra with Applications*, 2(2), 1995.

[Die97] R. Diestel. *Graph Theory*. Number 173 in Graduate Texts in Mathematics. Springer, 1997.

[Dij59] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[Fos95] I. Foster. *Designing and Building Parallel Programs*. Addison–Wesley, 1995.

[GHSJ96] S. K. S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson. A framework for generating distributed-memory parallel programs for block recursive algorithms. *Journal of Parallel and Distributed Computing*, 34(2):137–153, May 1996.

[Gib93] P. B. Gibbons. Asynchronous PRAM algorithms. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 22, pages 957–997. Morgan Kaufmann, 1993.

[GM84a] M. Gondran and M. Minoux. *Graphs and Algorithms*. Wiley—Interscience Series in Discrete Mathematics. John Wiley & Sons, 1984.

[GM84b] M. Gondran and M. Minoux. Linear algebra in dioids: A survey of recent results. *Annals of Discrete Mathematics*, 19:147–164, 1984.

[GMR] P. Gibbons, Y. Matias, and V. Ramachandran. Can a shared memory model serve as a bridging model for parallel computation? *Theory of Computing Systems*. To appear.

[GMR99] P. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, 28(2):733–769, April 1999.

[GMT88] H. Gazit, G. L. Miller, and Shang-Hua Teng. Optimal tree contraction in an EREW model. In S. K. Tewksbury, B. W. Dickinson, and S. C. Schwartz, editors, *Concurrent Computations: Algorithms, Architecture and Technology*, pages 139–156, 1988.

[Goo96] M. Goodrich. Communication-efficient parallel sorting. In *Proceedings of the 28th ACM STOC*, 1996.

[GPS90] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, March 1990.

[GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[GS96] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proceedings of the 8th ACM SPAA*, pages 223–232, 1996.

[GvdG96] B. Grayson and R. A. van de Geijn. A high performance parallel Strassen implementation. *Parallel Processing Letters*, 6(1):3–12, 1996.

[Had57] H. Hadwiger. *Vorlesungen über Inhalt, Oberfläche und Isoperimetrie*. Number 93 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1957.

[HJB] D. R. Helman, J. JáJá, and D. A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. *Journal of Experimental Algorithmics*. To appear.

[HK71] J. E. Hopcroft and L. R. Kerr. On minimizing the number of multiplications necessary for matrix multiplication. *SIAM Journal of Applied Mathematics*, 20(1):30–36, January 1971.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison–Wesley, 1992.

[JKS93] H. Jung, L. M. Kirousis, and P. Spirakis. Lower bounds and efficient algorithms for multiprocessor scheduling of directed acyclic graphs with communication delays. *Information and Computation*, 105(1):94–104, July 1993.

[JM95] D. B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. *Journal of Algorithms*, 19(3):383–401, November 1995.

[Joh77] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, January 1977.

[KHSJ95] B. Kumar, C.-H. Huang, P. Sadayappan, and R. W. Johnson. A tensor product formulation of Strassen's matrix multiplication algorithm with memory reduction. *Scientific Programming*, 4(4):275–289, 1995.

[KKT95] D. R. Karger, P. H. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–328, March 1995.

[KR90] R. M. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 17, pages 869–941. Elsevier, 1990.

[KRS90] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95–132, March 1990.

[KS96] J. Komlós and M. Simonovits. Szemerédi's Regularity Lemma and its applications in graph theory. Technical Report 96-10, DIMACS, 1996.

[LD90] S. Lakshmivarahan and S. K. Dhall. *Analysis and design of parallel algorithms: Arithmetic and matrix problems.* McGraw-Hill series in supercomputing and parallel processing. McGraw-Hill, 1990.

[LD94] S. Lakshmivarahan and S. K. Dhall. *Parallel computing using the prefix problem.* Oxford University Press, 1994.

[LLS+93] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Computing*, 19:1079–1103, October 1993.

[LM88] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988.

[LMR96] Zhiyong Li, P. H. Mills, and J. H. Reif. Models and resource metrics for parallel and distributed computation. *Parallel Algorithms and Applications*, 8:35–59, 1996.

[LW49] L. H. Loomis and H. Whitney. An inequality related to the isoperimetric inequality. *Bulletin of the AMS*, 55:961–962, 1949.

[McC93] W. F. McColl. General purpose parallel computing. In A. Gibbons and P. Spirakis, editors, *Lectures on parallel computation*, volume 4 of *Cambridge International Series on Parallel Computation*, chapter 14, pages 337–391. Cambridge University Press, 1993.

[McC95] 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.

[McC96a] W. F. McColl. Private communication, 1996.

[McC96b] W. F. McColl. A BSP realisation of Strassen's algorithm. In M. Kara, J. R. Davy, D. Goodeve, and J. Nash, editors, *Abstract Machine Models for Parallel and Distributed Computing*, pages 43–46. IOS Press, 1996.

[McC96c] W. F. McColl. Universal computing. In L. Bougé et al., editors, *Proceedings of Euro-Par '96 (I)*, volume 1123 of *Lecture Notes in Computer Science*, pages 25–36. Springer-Verlag, 1996.

[McC98] W. F. McColl. Foundations of time-critical scalable computing. In *Proceedings of the 15th IFIP World Computer Congress*. Osterreichische Computer Gesellschaft, 1998.

[Mit70] D. S. Mitrinović. *Analytic Inequalities*. Number 165 in Grundlehren der mathematischen Wissenschaften. Springer-Verlag, 1970.

[MMT95] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: a survey and synthesis. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, volume 2, pages 61–70. IEEE Press, 1995.

[Mod88] J. J. Modi. *Parallel Algorithms and Matrix Computation*. Oxford applied mathematics and computing science series. Clarendon Press, 1988.

[MP88] B. M. Maggs and S. A. Plotkin. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26(6):291–293, January 1988.

[MR85] G. L. Miller and J. F. Reif. Parallel tree contraction and its applications. In *Proceedings of the 26th IEEE FOCS*, pages 478–489, 1985.

[MT] W. F. McColl and A. Tiskin. Memory-efficient matrix multiplication in the BSP model. *Algorithmica*. To appear.

[Ort88] J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Frontiers of Computer Science. Plenum Press, 1988.

[Pat74] M. S. Paterson. Complexity of product and closure algorithms for matrices. In *Proceedings of the 2nd International Congress of Mathematicians*, pages 483–489, 1974.

[Pat93] M. S. Paterson. Private communication, 1993.

[PU87] C. H. Papadimitriou and J. D. Ullman. A communication-time tradeoff. *SIAM Journal of Computing*, 16(4):639–646, August 1987.

[PY90] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal of Computing*, 19(2):322–328, April 1990.

[Ram99] V. Ramachandran. A general purpose shared-memory model for parallel computation. In M. T. Heath, A. Ranade, and R. S. Schreiber, editors, *Algorithms for Parallel Processing*, volume 105 of *IMA Volumes in Mathematics and Applications*. Springer-Verlag, 1999.

[RM96] M. Reid-Miller. List ranking and list scan on the Cray C–90. *Journal of Computer and System Sciences*, 53(3):344–356, December 1996.

[RMMM93] M. Reid-Miller, G. L. Miller, and F. Modugno. List ranking and parallel tree contraction. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 3, pages 115–194. Morgan Kaufmann, 1993.

[Rot90] G. Rote. Path problems in graphs. *Computing Supplementum*, 7:155–189, 1990.

[Sch73] A. Schönhage. Unitäre Transformationen großer Matrizen. *Numerische Mathematik*, 20:409–417, 1973.

[Sib97] J. F. Sibeyn. Better trade-offs for parallel list ranking. In *Proceedings of 9th ACM SPAA*, pages 221–230, 1997.

[Sny98] L. Snyder. A ZPL programming guide (version 6.2). Technical report, University of Washington, January 1998.

[SS92] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[ST98] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[Tak98] T. Takaoka. Subcubic cost algorithms for the all pairs shortest path problem. *Algorithmica*, 20:309–318, 1998.

[TB95] A. Tridgell and R. P. Brent. A general-purpose parallel sorting algorithm. *International Journal of High-Speed Computing*, 7(2):285–301, 1995.

[Tis96] A. Tiskin. The bulk-synchronous parallel random access machine. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proceedings of Euro-Par '96 (II)*, volume 1124 of *Lecture Notes in Computer Science*, pages 327–338. Springer-Verlag, 1996.

[Tis98] A. Tiskin. The bulk-synchronous parallel random access machine. *Theoretical Computer Science*, 196(1–2):109–130, April 1998.

[Tol97] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal of Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[UY91] J. D. Ullman and M. Yannakakis. High-probability parallel transitive closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, February 1991.

[Val89] 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.

[Val90a] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[Val90b] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 18, pages 943–971. Elsevier, 1990.

[Zim81] U. Zimmermann. *Linear and Combinatorial Optimization in Ordered Algebraic Structures*, volume 10 of *Annals of Discrete Mathematics*. North-Holland, 1981.