národní
úložiště
šedé
literatury

Formalizing, Analyzing and Extending the Model of Bulk Synchronous Parallel Computer

Beran, Martin
2000

Dostupný z http://www.nusl.cz/ntk/nusl-33970

Dílo je chráněno podle autorského zákona č. 121/2000 Sb.

Tento dokument byl stažen z Národního úložiště šedé literatury (NUŠL).

Datum stažení: 03.05.2024

Další dokumenty můžete najít prostřednictvím vyhledávacího rozhraní nusl.cz .

INSTITUTE OF COMPUTER SCIENCE

ACADEMY OF SCIENCES OF THE CZECH REPUBLIC

Formalizing, Analyzing, and Extending the Model of Bulk Synchronous Parallel Computer

Martin Beran

Technical report No. 829

December 2000

Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2, 182 07 Prague 8, Czech Republic
phone: (+4202) 6884244 fax: (+4202) 8585789
e-mail: uivt@uivt.cas.cz

INSTITUTE OF COMPUTER SCIENCE

ACADEMY OF SCIENCES OF THE CZECH REPUBLIC

Formalizing, Analyzing, and Extending the Model of Bulk Synchronous Parallel Computer

Martin Beran[1] [2]

Technical report No. 829
December 2000

Abstract

The bulk-synchronous parallel (BSP) computer — introduced by Valiant in [71] — is a paradigmatic example of a bridging model of parallel computation. It is a parametrized model that realistically describes performance of existing parallel computers while retaining independence on concrete architectures. We analyze the BSP model from the complexity theory point of view. Instead of being focused on algorithms, we rather study relations between the BSP machines and classes of sequential and parallel models of computations. It turns out that computational power of the BSP depends strongly on the values of its parameters. We will show that BSP computers with slow communication are roughly equivalent (up to a polynomial factor) to sequential models like Turing machines or RAMs. On the other hand, very fast communication makes a BSP as fast as the massively parallel machines, e.g., a PRAM. The standard BSP lacks any sense of communication locality. Therefore we define an extended model: decomposable BSP (dBSP). The processors of the dBSP computer can be dynamically partitioned into clusters. In the same cluster, the processors can communicate faster than in the non-partitioned machine, but communication among processors from different clusters is forbidden. To illustrate the power of dBSP, we will show how elementary BSP algorithms (e.g., broadcasting, prefix sums, matrix multiplication, and simulation of cellular automata) are sped up by their adaptation to the decomposable BSP model. Issues of simulation of dBSP on BSP and of mutual simulations of several versions of the dBSP computers are discussed. Finally, we analyze how the parameters of the dBSP model influence its membership in the three basic machine classes (of sequential, massively parallel, and weak parallel machines).

Keywords

models of computation, parallel computers, parallel algorithms, BSP, dBSP, machine classes, pipelining

[1]Faculty of Mathematics and Physics Charles University, Prague; phone:+4202 2191 4205, fax:+4202 57533961, email:beran@ss1000.ms.mff.cuni.cz

[2]This research was partially supported by GA ČR grant No. 201/00/1489.

Contents

1 Introduction **5**

2 Preliminaries **9**
2.1 Basic Definitions 10
2.2 First Class Models 12
2.3 Second Class Models 14
2.4 Weak Parallel Models 20

3 Analysis of the BSP Model **29**
3.1 Definition of BSP 29
3.2 Elementary Features of the BSP Model 33
3.3 BSP and Machine Classes 36
3.3.1 BSP Computers in the First Machine Class 36
3.3.2 BSP in the Second Machine Class 38
3.3.3 Outside the First and the Second Machine Classes 40
3.3.4 Relation to Weak Parallel Machines 41

4 BSP with Communication Locality: Decomposable BSP **45**
4.1 Definition of a Decomposable BSP Computer 46
4.2 Algorithms for dBSP Computers 48
4.2.1 Tree Computations 50
4.2.2 Broadcasting and Aggregation of n Elements 53
4.2.3 Dense Matrix Multiplication 57
4.2.4 Simulation of Cellular Automata 62
4.3 Simulations of BSP and dBSP Computers 66
4.4 Variants of dBSP Computers 68
4.4.1 Renumbering dBSP 68
4.4.2 Asynchronous dBSP 73
4.5 Membership of dBSP in Machine Classes 75
4.5.1 Membership of dBSP in $\mathcal{C}_1$ 75
4.5.2 Membership of dBSP in $\mathcal{C}_2$ 75
4.5.3 Outside $\mathcal{C}_1$ and $\mathcal{C}_2$ 76
4.5.4 Weak Parallel Machines and dBSP 77
4.6 Decomposable BSP and the Real World 81

5 Conclusion **83**

Chapter 1

Introduction

The primary aim of this thesis is to analyze the computational efficiency of Bulk Synchronous Parallel (BSP) machines and compare them to other models of computation (both sequential and parallel). We will not do it from the point of view of real, existing parallel computers and "practical" algorithms. Rather, we will use the standard framework of computational complexity theory, such as formal machine models, machine classes, and simulation between models. We will identify some problems in BSP concerning its lack of communication locality and thus unsuitability for pipelined computations. The secondary goal is to solve these problems by designing and analyzing an extended model — Decomposable BSP (dBSP).

First, we present a short overview of present state in the area of machine models, its main problems, and goals we would like to achieve.

The present state. Formal models of computation are necessary prerequisites for building a theory of computational complexity. In the area of sequential computing, there are two widely accepted formal models of computers: the Turing machine and the random access machine (RAM). Especially the RAM captures the essential features of existing sequential computers very well. Algorithms can be designed and their complexity can be analyzed using the RAM model. When such an algorithm is then implemented on a real machine, its actual performance asymptotically matches the estimated theoretical values. Both the RAM and the Turing machine are precisely defined, allowing not only design and analysis of algorithms, but also formal reasoning about complexity classes and lower bound proofs.

The situation in the field of parallel computation is much more complicated and less satisfactory. Historically, a large gap emerged between really built parallel computers and the formal models of massively parallel machines, such as SIMDAG [32] or PRAM. The reader interested in PRAM algorithms and complexity theory can find more information in [20, 41, 44, 76].

Building shared memory machines reasonably corresponding to the PRAM model is complicated because the model underestimates the real cost of accessing the global memory by many processors simultaneously. A typical real shared memory parallel computer has only up to several tens of processors. On the other hand, PRAM algorithms often use the number of processors polynomial or even exponential in the size of the input. Using shared memory, any two processors can exchange information in a constant time. Bilardi and Preparata [10] and Wiedermann [80] analyze physical feasibility of parallel machines. The basic laws of nature cause that the full power of PRAM cannot be implemented by a physically feasible machine.

To allow an efficient and scalable implementation, the PRAM model must be restricted. Possible restrictions are either to limit the number of processors, or to charge more for operations with the global memory. Examples of the latter approach are the delay PRAM [56] and the QSM (Queuing Shared-Memory) [31]. A different approach is used in the PADAM model [19] which has its global memory partitioned into modules and each processor can access only some of the modules.

Massively parallel computers with hundreds or thousands of processors usually possess only distributed memory. Each processor has its own local memory and the processors are connected by a communication network. The communication is done by message passing. Theoretical models of distributed memory computers were developed for design and performance modeling of algorithms. The parallel computer is viewed as a set of RAMs with additional capability of sending and receiving messages to/from the interconnection network. The topology and routing performance of the network strongly influence behavior of a parallel algorithm. There are many possible architectures, e.g., bus, mesh, torus, butterfly, hypercube, or crossbar [36, 57, 58, 63]. Algorithms developed with a concrete model in mind run efficiently on real computers and their real performance matches the theoretical estimations, but they are usually non-portable to other architectures.

Given the problems with PRAM on one side and message passing networks on the other, research was aimed at finding a *bridging model.* According to Valiant [71], such a model would constitute "an efficient bridge between software and hardware: high-level languages can be efficiently compiled on to this model; yet it can be efficiently implemented in hardware." A good candidate for a bridging model should describe the underlying parallel machine by only few parameters. Algorithms are then developed and optimized with respect to the model and its parameters, without any knowledge of the underlying architecture. An efficient implementation of the model should be possible for a large class of really existing parallel computers. The *Bulk Synchronous Parallel (BSP)* model was defined in the paper [71] as a possible bridging model. Many BSP algorithms were developed and analyzed [6, 17, 29, 30, 33, 42, 49, 50, 51, 52, 64, 65, 71]. There are also some implementations of the BSP model, e.g., [11, 16, 26, 37, 38]. Other bridging models are LogP [9, 23], CTA/PA [67], BSP* [2, 3], C^3 [34], BSPRAM [53, 69, 70] CGM [14, 15], and QSM [31, 60]. For survey of parallel machine models, see [24, 47].

Our goals. Slot and van Emde Boas defined the *first machine class* $\mathcal{C}_1$ [66] as a class containing the Turing machine and other sequential models [1, 12, 22, 35, 39, 40], such that some basic complexity classes like P, $PSPACE$, and $LOGSPACE$ remain invariant for all models belonging to the first class. Similarly, the second machine class $\mathcal{C}_2$ was introduced by van Emde Boas [73] as the class of parallel (PRAM, k-PRAM [62], uniform logical circuits [13]), vector (vector machine [59], array processing machine [74]), and non-deterministic (alternating Turing machine [18]) models. An interesting fact is that the RAM with multiplication and division and uniform cost is also a second class device [8]. Wiedermann introduced the class $\mathcal{C}_{\text{weak}}$ of weak parallel machines [78, 79]. Members of $\mathcal{C}_{\text{weak}}$ are parallel devices, but they can achieve smaller degree of parallelism in comparison with the second class computers. Weak parallel machines are suitable especially for pipelined computations. An overview of machine models, classes, and simulations is presented in [72]. How is the BSP model related to the machine classes? Our goal is to find an answer to this question. We will show that it depends heavily on the values of parameters p, g, l that characterize performance of the BSP model.

By its nature, BSP is a suitable model for algorithms which need all-to-all communication among processors. Many important algorithms do not need general communication abilities (at least not during all parts of computation). They manifest *communication locality.* E.g., numerical iterative algorithms evaluate a new value of a matrix element using values of its immediate neighbors in the previous step. Some global termination criterion is checked only from time to time. When such an algorithm is implemented on a real parallel computer with a communication network, its performance can be improved by putting often communicating processes on nearby processors, because communication cost depends on the distance between the communicating processors. In this thesis, we will try to formulate an answer to the question whether it is possible to retain portability of a bridging model and benefit from locality at the same time. We will define the *decomposable BSP model (dBSP)* as an extension to the BSP. We will present algorithms which show a substantial speedup on the dBSP in comparison with the ordinary BSP. This work is partially inspired by the idea of recursive divisibility [46, 55, 56] and by the reconfigurable parallel machine models, e.g., meshes with buses [4, 45, 68]. Some previous attempts to extend the BSP model by a notion of locality include the E-BSP [43] and BSP2 [48]. The cost model of E-BSP is not compatible with the original BSP and the BSP2 allows only two fixed levels of locality. On the other hand, the decomposable BSP can be adaptively reconfigured according to the needs of the algorithm. If the additional dBSP features are not used, the dBSP is equivalent to a BSP and its time complexity is exactly the same as for the BSP model.

Structure of the thesis. The text consists of three main parts. Chapter 2 contains an overview of basic definitions and facts which will be used later. They include Turing machines, RAMs, PRAMs, machine classes, and parallel Turing machines. The new results are the proofs that, in order to belong to the class of weak parallel machines, the computational power of the pipelined parallel Turing machine must be restricted. Two such restrictions — limited and strictly pipelined parallel Turing machines — will be presented. In Chapter 3, we will study the BSP model. We will see that traditional informal definition of the BSP computer is not sufficient for our purposes. Hence we will start by presenting a more formal definition. Then we will analyze membership of the BSP in the first and the second machine class and in the class of weak parallel machines. Chapter 4 is devoted to decomposable BSP computers. We will define the model first. Then we will describe some elementary algorithms and compare their performance on BSP and dBSP machines. We will also show how relations to machine classes are influenced by decomposability. The results presented here were previously published in two papers by the author of this thesis. The contents of Chapter 3 are based on the paper [5]. The decomposable BSP computers were introduced in [7].

Chapter 2

Preliminaries

We assume that the reader is familiar with fundamental concepts of complexity theory and with standard sequential and parallel machine models. The (non-exhaustive) list of prerequisites contains: Turing machines (TM) with various numbers of heads and tapes, Random Access Machines (RAM), Parallel Random Access Machines (PRAM), determinism, non-determinism, time $T(n)$ and space $S(n)$ complexity measures, and basic complexity classes (especially LOGSPACE, P, NP, PSPACE). A preliminary knowledge about the Bulk Synchronous Parallel (BSP) model would be helpful, but is not necessary. A good overview and references to additional sources of information can be found in [51, 52, 72].

In this chapter, we recall some well known definitions and results. Formal definitions are necessary to allow rigorous reasoning needed for proving upper and lower bounds on complexity of algorithms. Our main goal here is to clarify which of different versions found in literature we use. To achieve consistency of presentation, no specific version of some definitions (e.g., RAM and PRAM) is directly taken from literature, but rather features from typical definitions are combined. For these models, the respective proofs, e.g., membership of RAM in the first and PRAM in the second machine class, are also newly formulated, although the results are already known. The RAM and PRAM models are described in detail, because there are many versions of them, varying in such important characteristics like the instruction set or complexity measures. Hence, it is necessary to carefully define which models we use. On the other hand, the other models, like various modifications of the Turing machine, are only informally described. The well-known proofs of membership of the RAM in the first class and of the PRAM in the second class are presented in detail, because the respective simulation algorithms will be referenced in subsequent sections. The respective algorithms will be used when proving membership of the BSP and the dBSP computers in particular machine classes.

The only new results in this chapter are in Section 2.4. It is pointed to a bug in the original definition of the pipelined parallel Turing machine (PPTM) [79] that prevents its membership in the class of weak parallel machines. Two definitions of the PPTM with restricted computational power are proposed, namely the limited PPTM and the strictly pipelined PTM. Membership in the class of weak parallel machines is proved for both of them.

If not explicitly stated otherwise, all logarithms are with base 2. Throughout this thesis, we assume that all bounds on time, space, work, number of processors, and word size are given by functions constructible in the same time, space, work, number of processors, and word size

as given by the value of the function itself. Thus, an algorithm can always evaluate (constant times) its own bounding function and its asymptotic complexity will be still bounded by the same bounding function.

We distinguish between *computational steps* and *time units*. A processor of a computer executes one instruction per step which can cost one or more time units. Under *uniform cost*, each computational step takes exactly one time unit. When *logarithmic cost* is used, different instructions or the same instruction with different operands can take different numbers of time units. The number of time units per a computational step, i.e., per instruction, is equal to the bit length of the largest argument of that instruction. We use *logarithmic cost* throughout this thesis except for some parts where uniform cost is explicitly mentioned.

2.1 Basic Definitions

For our purposes, as usually, constant factors in complexity analysis will be of no importance. Therefore the standard "O" notation will be used throughout this thesis. Even weaker polynomial equivalence is sufficient to establish membership in machine classes.

Definition 2.1 *Given functions f and g, we use the following notation:*

1. $f = O(g)$ *iff*[1] $\exists k > 0\ \exists n_0\ \forall n \geq n_0\ : f(n) \leq kg(n)$,
2. $f = \Omega(g)$ *iff* $\exists k > 0\ \exists n_0\ \forall n \geq n_0\ : f(n) \geq kg(n)$,
3. $f = \Theta(g)$ *iff* $f = O(g) \wedge f = \Omega(g)$ *; such functions are said to be* linearly equivalent
4. $f = o(g)$ *iff* $\limsup_{n\to\infty} f(n)/g(n) = 0$,
5. $f = \omega(g)$ *iff* $\limsup_{n\to\infty} g(n)/f(n) = 0$.

Definition 2.2 *Functions f and g are* polynomially equivalent *iff* $\exists k \geq 0\ \exists l \geq 0\ : f(n) = O(g^k(n))$ *and* $g(n) = O(f^l(n))$.

There are some equivalences among various computational models that allow to assign the models into a few machine classes. Polynomial equivalence of time complexity classes and linear equivalence of space complexity classes ensure that fundamental complexity classes, such as P and LOGSPACE, are invariant to a choice of a particular model from the same class. This property is captured in the next two definitions.

Definition 2.3 *The* first machine class $\mathcal{C}_1$ *contains deterministic one-tape, one-head Turing machines (DTMs) and all machines which are polynomially equivalent in time and linearly equivalent in space to DTMs. Time and space bounds must be reached simultaneously by the same simulation algorithm.*

Definition 2.4 *The* second machine class $\mathcal{C}_2$ *is the class of computational devices with their time complexity polynomially equivalent to the space complexity of a DTM.*

---
[1]abbreviation of "if and only if"

The first class contains common sequential machine models, such as Turing machines and RAMs (with the logarithmic cost function). Members of the second class are either massively parallel machines with many processors (PRAM), or machines allowing large data processing in a single step (RAM with the uniform cost function), or those having a modified acceptance rule, which generates some kind of parallelism (alternating Turing machine). Note that unless P $\neq$ PSPACE is known, we are not certain whether the first and the second class differ. An interested reader can find more information about machine models, their mutual simulations, and machine classes in [72].

Parallel computers bring new complexity measures: the *number of processors* used during a computation, and the *work* which is the sum of all computational steps performed by all processors. If the load of all processors is more or less balanced, it is roughly equal to the product of the time and the number of processors. More precisely, the work is $\sum_{t=1}^{T(n)} p_t$, where p_t is the number of processors active at the time unit t.

Pipelining is an efficient way of processing a sequence of many instances of the same problem on a parallel computer. The time complexity of computations of individual instances is not improved, but the whole sequence can be processed much faster than by repeatedly using a sequential algorithm on each individual instance of the problem. A new complexity measure — *period* — is introduced to describe performance of pipelining. The class of weak parallel machines is then defined as a class of computers which can perform fast pipelined computations [79].

Definition 2.5 Period $P(n)$ *of a computation over a sequence of inputs of the same size* n *is the upper bound on time between beginnings of reading two subsequent inputs of the sequence or between ends of writing two subsequent outputs.*

Definition 2.6 *The* class of weak parallel machines $\mathcal{C}_{\text{weak}}$ *contains machines with their periods polynomially equivalent to the space complexity of a DTM.*

To prove membership of a machine $\mathcal{M}$ in a machine class $\mathcal{C}$, we can use the respective definition of $\mathcal{C}$ directly. Other possibility is to choose a model $\widehat{\mathcal{M}}$ known to belong to $\mathcal{C}$, design a mutual simulation between $\mathcal{M}$ and $\widehat{\mathcal{M}}$, and apply the following lemma.

Lemma 2.7

1. *Machine model* $\mathcal{M}$ *is a member of the first class, iff there is model* $\widehat{\mathcal{M}} \in \mathcal{C}_1$ *such that* $\mathcal{M}$ *and* $\widehat{\mathcal{M}}$ *can be mutually simulated with polynomially equivalent time complexities and linearly equivalent space complexities.*

2. *Machine model* $\mathcal{M}$ *is a member of the second class, iff there is model* $\widehat{\mathcal{M}} \in \mathcal{C}_2$ *such that* $\mathcal{M}$ *and* $\widehat{\mathcal{M}}$ *can be mutually simulated with polynomially equivalent time complexities.*

3. *Machine model* $\mathcal{M}$ *is a member of the weak parallel class, iff there is model* $\widehat{\mathcal{M}} \in \mathcal{C}_{\text{weak}}$ *such that* $\mathcal{M}$ *and* $\widehat{\mathcal{M}}$ *can be mutually simulated with polynomially equivalent periods.*

Proof. Follows immediately from definitions of machine classes. □

2.2 First Class Models

A one-tape, one-head deterministic Turing machine is in the first class by definition. Any multi-tape, multi-head deterministic Turing machine is also a member of $\mathcal{C}_1$. This fact follows from well known simulation results for Turing machines [12, 35, 39, 40]. Other widely used sequential computer model is the deterministic random access machine. It belongs to the first class too, but its time and space complexity measures must be defined with some caution [1, 22]. We use the definition in which the address of a register contributes to space occupied by the register. Any RAM can be then easily simulated by a Turing machine in linear space. See the discussion of problems in simulations of RAMs on Turing machines in [72, 77]. Further, we require that both the Turing machine and the RAM must always read their whole input. Hence their time complexity is always at least n. The Turing machine is off-line, i.e., the input symbols are placed on a separated input tape where they can be accessed (repeatedly) by the machine as needed during the whole computation. The input tape of length n contributes by an additional term of $\log n$ to the machine's space complexity, because a number between 0 and $n-1$ can be encoded by the head position on the input tape.

Definition 2.8 (Random access machine — RAM)
A random access machine has a memory consisting of an unbounded number of registers indexed by non-negative integers. Each register is capable to store an arbitrary integer. The computation is controlled by a fixed program — a finite sequence of labeled instructions of form:

- `x:=y` *— assignment*
- `z:=x+y`, `z:=x-y`, `z:=x*y`, `z:=x/y` *(integer division) — arithmetic operations*
- `goto` *label — unconditional jump*
- `if x>=0 then goto` *label — conditional jump*
- `halt` *— end of computation*

The values `x`, `y`, *and* `z` *in the instructions can be*

- n *— a constant*
- r_n *— content of the n-th register (direct addressing)*
- r_{r_n} *— content of the r_n-th register (indirect addressing)*

In the beginning of a computation, the input consisting of r_{in} values (occupying n bits in total) is stored in registers $0, \ldots, r_{\text{in}} - 1$. All remaining registers are zero. The computation starts at the first instruction, performs one instruction at one step and continues at the next instruction (if a jump is not performed). When `halt` *is encountered the computation is terminated and registers $r_{\text{in}}, \ldots, r_{\text{in}} + r_{\text{out}} - 1$ contain the output. During the computation, the whole input must be read, i.e., all input registers accessed.*

The time complexity *of a RAM computation on input x is defined as*

$$T^{\mathrm{RAM}}(x) = \sum_{i=1}^{I} |b_i| \ ,$$

where I is the number of instructions performed until **halt** *and $|b_i|$ is the number of bits of the operand with largest absolute value involved in the i-th step. Both register address and value count as operands.*

The space complexity *of a RAM computation is*

$$S^{\mathrm{RAM}}(x) = \sum_i (|i| + |r_i|) \text{ , for each used register } i$$

$|i|$ is the length of the binary form of address i and $|r_i|$ is the maximum number of bits stored in the i-th register during the computation. Input registers which are not changed during computation and output registers which are only written but not read are not counted to the space complexity[2]*.*

We do not use input and output tapes for a RAM. Instead both its input and output are placed in registers. Hence no input/output instructions are needed and we need not bother with conversions between registers and tape symbols. As it is usual, we also define for any input of a given size $T^{\mathrm{RAM}}(n) = \max_{x:|x|\leq n}\{T^{\mathrm{RAM}}(x)\}$ and $S^{\mathrm{RAM}}(n) = \max_{x:|x|\leq n}\{S^{\mathrm{RAM}}(x)\}$. We use the *logarithmic cost* functions for both time and space complexity, i.e., the number of bits involved in a computation is taken into account. Other possibility is to use the *uniform cost*, i.e., counting the number of steps performed and the number of registers used. The uniform cost is often used when analyzing complexity of algorithms, but then the *word size* $\max_i |r_i|$ must be suitably bounded, usually by $O(\log n)$, to preserve membership in the first class. Using uniform time cost with unbounded word size causes RAM to belong into the second class. Moreover, the space complexity of any algorithm according to the uniform space cost is $S(n) \leq 2$, because a two-counter machine is a universal computational device [54]. We will adopt the common practice of using the uniform cost only for bounded word size algorithms when it simplifies the results. But if not explicitly said otherwise, the *logarithmic cost* is always used. The uniform and the logarithmic cost time complexities ($T^{\mathrm{RAM}}_{\mathrm{unit}}$ and $T^{\mathrm{RAM}}_{\mathrm{log}}$, respectively), are related as follows:

Lemma 2.9 *For input x and for an algorithm with word size $W(x)$ holds*

$$T^{\mathrm{RAM}}_{\mathrm{unit}}(x) \leq T^{\mathrm{RAM}}_{\mathrm{log}}(x) \leq T^{\mathrm{RAM}}_{\mathrm{unit}}(x)W(x) \text{ .}$$

Proof. The claim follows directly from the definition and from the fact that $W(x)$ is an upper bound on the number of bits of the largest operand used during the computation. □

The logarithmic time cost induces an upper bound on the word size. The time complexity of an algorithm limits the number of bits that can be placed in a register, and thus also the highest addressable register.

Lemma 2.10 *The largest number stored in a register and the highest address used during a $T(n)$ time bounded RAM computation*[3] *is at most $2^{O(T(n))}$.*

Proof. All the n input bits may be stored in one register which then contains a number less or equal to $2^n \leq 2^{T(n)}$ according to the assumption $T(n) \geq n$. Execution of any instruction which involved a number or address greater than $2^{O(T(n))}$ would consume more than $T(n)$ time units. □

[2]This is a necessary condition for sublinear (e.g., logarithmic) space complexity.

[3]Recall that we use logarithmic cost.

The time and space complexities of an algorithm are not independent, but there can be an exponential gap between them. The fact that space is upper bounded by time is used several times in the proof that RAM is a member of the first machine class.

Lemma 2.11 *On both a Turing machine and a RAM, time and space complexities are interrelated by the inequality*

$$S(n) \le T(n) \le 2^{O(S(n))} .$$

Proof. The first inequality follows from the fact that in $T(n)$ steps a Turing machine cannot use more than $T(n)$ cells and a RAM cannot allocate more than $T(n)$ bits in its registers. After $2^{O(S(n))}$ steps, all possible configurations in space $S(n)$ are exhausted and the machine has either to stop or to cycle forever. □

Theorem 2.12 $RAM \in \mathcal{C}_1$.

Proof.

Simulation of a RAM on a Turing machine:

Tuples $\langle i, r_i \rangle$ for the currently used RAM registers are written on the TM tape in binary encoding. The TM simulates one RAM step by traversing the tape, finding the relevant registers, simulating one RAM instruction and updating the tape. Clearly, the simulation runs in space $S^{\mathrm{TM}}(n) = O(S^{\mathrm{RAM}}(n))$ and time $T^{\mathrm{TM}}(n) = O\Big(T^{\mathrm{RAM}}(n)(S^{\mathrm{RAM}}(n))^2 + nT^{\mathrm{RAM}}(n)\Big) \le O\Big((T^{\mathrm{RAM}}(n))^3\Big)$ — each of $T^{\mathrm{RAM}}(n)$ steps requires in the worst case visiting each cell of the tape, moreover multiplication and division of large numbers introduce additional factor of $S^{\mathrm{RAM}}(n)$ if multiplication and division are done in $|r_i|$ steps on a RAM and $|r_i|^2$ on a TM. The term $nT^{\mathrm{RAM}}(n)$ corresponds to reading of an input register, because the TM must, in the worst case, traverse the whole input tape. Nevertheless, the requirement that the RAM must read the complete input yields $T^{\mathrm{RAM}}(n) \ge n$.

Simulation of a TM on a RAM:

The RAM utilizes $S^{\mathrm{TM}}(n)/\log S^{\mathrm{TM}}(n)$ registers and stores $O(\log S^{\mathrm{TM}}(n))$ tape cells (bits) in each register. Additionally a register contains $\log(S^{\mathrm{TM}}(n)/\log S^{\mathrm{TM}}(n)) = O(\log S^{\mathrm{TM}}(n))$ address bits. The total space needed is therefore $S^{\mathrm{TM}}(n) = O(S^{\mathrm{TM}}(n))$. One TM step is simulated by extracting a constant number of bits containing encoded neighborhood of the TM head from one RAM register, simulating the instruction and packing the bits back to the register. Bit operations are performed by multiplication and division by a power of 2 in time $O(\log S^{\mathrm{TM}}(n) + \log n) \le O(\log T^{\mathrm{TM}}(n))$. The term $\log n$ corresponds to reading an input cell. The TM must read the whole input, hence $T^{\mathrm{TM}}(n) \ge n$. Therefore $T^{\mathrm{RAM}}(n) = O(T^{\mathrm{TM}}(n) \log T^{\mathrm{TM}}(n))$.

Both simulations run with a polynomial time overhead and a linear space overhead. □

2.3 Second Class Models

There are many machine models in the second class, e.g., the alternating Turing machine, RAM with uniform time cost, and various versions of the Parallel Random Access Machine

(PRAM) [32, 41, 44]. We are interested in formalization of the Bulk Synchronous Parallel (BSP) model and in comparison of BSP with the second class models. As BSP and PRAM processors are similar in their functionality, we will use PRAM as a representative of the second class. There are many variants of the PRAM. There are SIMD PRAMs which execute the same instruction in all processors, and SPMD PRAMs with a single common program, but the processors can execute different instructions at the same time. PRAMs differ also in ways how they resolve read and write conflicts. Some possibilities are exclusive-read exclusive-write or concurrent-read concurrent-write. Fortunately, these differences do not affect membership of a PRAM in the second class. PRAM algorithms and a list of some other second class models can be found in [44, 72].

Definition 2.13 (SIMD CROW PRAM)
A Single Instruction Multiple Data Concurrent-Read Owner-Write Parallel Random Access Machine *consists of $p = p(n)$ processors which are indexed $0, \dots, p-1$. A processor has its index stored in special register* pid. *Every processor has a local memory (accessible only by this single processor) with an unbounded number of registers (like a RAM). The processors share the program and the program counter, i.e., at every step, they execute synchronously the same instruction. In addition to the local memories, there is a global memory with the CROW property. It means that any number of processors can read from any register simultaneously, but the i-th processor can write only into the single register it owns, i.e., the register with index i. The r_{in} input values are stored in global registers $0, \dots, r_{\text{in}}$, all the other global and local registers are initially set to zero. The whole input must be read during the computation. The valid program instructions are*

- ***x:=y**, **z:=x+y**, **z:=x-y**, **z:=x*y**, **z:=x/y** (integer division), **goto** label, **halt** — the same as RAM instructions, **x**, **y**, and **z** may be constants and directly or indirectly indexed local and global registers*

- ***x:=pid** — obtaining the index of the processor which executes this instruction*

- ***activate(p)** — activate the processor with index **p***

- ***if z>0 then x:=y** — conditional assignment*

- ***if all_zero(z) goto** label — conditional jump if each processor has 0 in its local register **z***

*In the beginning of a computation on input x (stored in the initial part of the global memory), only up to $O(|x|^k)$ processors are active for some constant $k \geq 0$. Originally inactive processor i starts computation when some other active processor executes the instruction **activate(i)**. When the instruction **halt** is executed, the computation is terminated and the output can be found in global registers $r_{\text{in}}, \dots, r_{\text{in}} + r_{\text{out}} - 1$.*

The time complexity *of an I-steps long computation on input x is*

$$T^{\text{PRAM}}(x) = \sum_{i=1}^{I} \max_{0 \leq j \leq p-1} \{|b_{i,j}|\} ,$$

where $|b_{i,j}|$ is the number of bits of the largest operand of the i-th step at processor j.

The space complexity *is defined as*

$$S^{\mathrm{PRAM}}(x) = \sum_{i=0}^{p-1} \left(|i| + |g_i| + \sum_j \left(|j| + |l_{i,j}| \right) \right) ,$$

where $|g_i|$ is the maximum number of bits stored during the computation in register i of the global memory, j goes over all used local registers in processor i and $|l_{i,j}|$ is the maximum number of bits stored during the computation in the j-th local register of processor i. Input registers which are not changed during computation and output registers which are only written but not read are not counted to the space complexity.

The input can be read in parallel, hence the time complexity can be less than n. Nevertheless, the requirement to read the whole input induces the lower bound $T(n) \geq \log n$.

Lemma 2.14 *The lower bound on the time complexity of a PRAM computation is $T(n) \geq \log n$.*

Proof. Let the n input bits be stored in r registers. Then at least one register contains at least n/r bits. Parallel reading of the whole input, i.e., accessing all r input registers, requires time $\Omega(\log r + \log(n/r)) = \Omega(\log n)$. □

The limit of the largest possible address and content of a register from Lemma 2.10 is valid for a PRAM as well. The number of active processors used in a PRAM computation is limited by the time complexity of the `activate(p)` instruction. The space complexity is limited by the cost of memory accesses due to the logarithmic cost.

Lemma 2.15 *For the maximum number of processors $p_{\max}$ used in the first T time units of a PRAM computation on an input of size n, it holds*

$$p_{\max} \leq 2^{O(\log n + T)} .$$

Proof. In the beginning of the computation there are up to $p_{\mathrm{init}} \leq O(n^k) \leq 2^{O(\log n)}$ processors. The logarithmic time cost puts an upper bound on the value of the argument given to the `activate` instruction, because this instruction must be finished in time T: $\log p_{\mathrm{activate}} \leq T$ yields $p_{\mathrm{activate}} \leq 2^{O(T)}$. □

Lemma 2.16 *Time and space complexities of a PRAM computer are related by the inequality*

$$S(n) \leq pT(n) \leq 2^{O(T(n))} .$$

Proof. During $T(n)$ time units, a single PRAM processor cannot allocate more than $T(n)$ bits in its local registers and in the global memory. See also the proof of Lemma 2.11.

Combination of the first part of this lemma with Lemma 2.15 yields $S(n) \leq pT(n) \leq 2^{O(T(n))}T(n) \leq 2^{O(T(n)+\log T(n))} \leq 2^{O(T(n))}$. □

The previous three lemmas hold not only for a SIMD CROW PRAM, but also for all variants of PRAM which will be defined later in this section.

Now we prove that any SIMD CROW PRAM is a member of the second machine class. Then we show that other PRAMs belong to the second class as well.

Theorem 2.17 *SIMD CROW PRAM* $\in \mathcal{C}_2$.

Proof.

Simulation of a PRAM on a TM:

We design a nondeterministic sequential algorithm which simulates a PRAM computation in polynomial space. Let us assume $T^{\text{PRAM}}(n) = \Omega(\log n)$. The maximum number of processors will be, according to Lemma 2.15, $p \leq 2^{O(T)}$. The global memory and the local memories of all processors cannot be stored on the TM tape, because it may be much larger than polynomial in $T^{\text{PRAM}}(n)$. Therefore contents of individual registers are individually recursively evaluated. The algorithm runs as follows:

1. Guess and write down the sequence of instructions executed ... space $S_1^{\text{TM}} = O(T^{\text{PRAM}})$.
2. Guess and write down the output ... space $S_2^{\text{TM}} = O(T^{\text{PRAM}})$.
3. Let us denote `instr(t)` the instruction executed in step t. We define function `mem(t,p,m)` which returns the content of register (p, m) — local register `m` of processor p (or the global register owned by processor p if $m = -1$) in step t.

   function `mem(t,p,m)`
       if `instr(t)` does not change `(p,m)` **then** return `mem(t-1,p,m)`;
       if `instr(t)`=(`if z>0 then x:=y`) & `mem(t-1,p,z)`>0 **then**
           return `mem(t-1,p,y)`;
       if `instr(t)`=(`x:=y`) **then** return `mem(t-1,p,y)`;
       if `instr(t)`=(`z:=x`$\circ$`y`) **then**
           return `mem(t-1,p,x)`$\circ$`mem(t-1,p,y)`, where $\circ \in \{+, -, *, /\}$
   end

   The output guessed in step 2 is checked by calling `mem(`T^{PRAM}`,o,-1)` for each output register o. The recursion depth is $S_{3'}^{\text{TM}} = O(T^{\text{PRAM}})$ and we need to store up to $S_{3''}^{\text{TM}} = O(T^{\text{PRAM}})$ bits per recursion level. Thus we get $S_3^{\text{TM}} = S_{3'}^{\text{TM}}.S_{3''}^{\text{TM}} = O((T^{\text{PRAM}})^2)$ for the space complexity of function `mem(t,p,m)`.
4. Check the sequence of instructions guessed in step 1. Given the label (index in the program) `instr(t)` of the instruction executed in the t-th step, check the label of the next executed instruction `instr(t+1)`.
   (a) `instr(0)`$= 0$,
   (b) if `instr(t)` is not a jump then `instr(t+1)`=`instr(t)`$+1$,
   (c) if `instr(t)`=(`goto l`) then `instr(t+1)`$= l$,
   (d) if `instr(t)`=(`if all_zero(z) goto l`) then `instr(t+1)`$= l$ if for each processor p, `mem(t,p,z)`$= 0$, otherwise `instr(t+1)`=`instr(t)`$+1$.

   Checking of the instructions needs a counter from 0 to T^{PRAM} plus a space for function `mem(t,p,z)` ... $S_4^{\text{TM}} = O(\log T^{\text{PRAM}} + S_3^{\text{TM}}) = O(S_3^{\text{TM}})$.

The total space needed by the simulation algorithm is $S_{\text{nondet}}^{\text{TM}} = S_1^{\text{TM}} + S_2^{\text{TM}} + S_3^{\text{TM}} + S_4^{\text{TM}} = O((T^{\text{PRAM}})^2)$. Now we transform our nondeterministic algorithm into a deterministic one using Savitch's theorem [61]. The space complexity of the deterministic algorithm is $S^{\text{TM}} = O((S_{\text{nondet}}^{\text{TM}})^2) = O((T^{\text{PRAM}})^4)$.

Simulation of a TM on a PRAM:

> There are $C = 2^{O(S^{\mathrm{TM}})}$ possible configurations in space S^{TM}. Reachability of the configurations defines a graph with vertices corresponding to individual configurations and edges representing possible transitions between configurations. We construct $C \times C$ adjacency matrix M of this graph, such that $M_{i,j} = 1$ if configuration j is reachable from configuration i in a single step of the Turing machine and $M_{i,j} = 0$ otherwise. We use a PRAM with $p = C^3$ processors and $\log C = O(S^{\mathrm{TM}})$ times repeat squaring of matrix M to compute the transitive closure of M. The TM is deterministic, therefore for each initial configuration i' there is single terminal configuration j'. Configuration j' can be easily found among all configurations reachable from i'. One computation of the square of the matrix and also the final search for j' take time $O(\log^2 C) = O((S^{\mathrm{TM}})^2)$. The total simulation time is then $T^{\mathrm{PRAM}} = O((S^{\mathrm{TM}})^3)$.

The above simulations establish a polynomial equivalence between the sequential space and the parallel time. Thus a PRAM belongs to the second machine class. □

Definition 2.18 (CROW PRAM)
A Concurrent-Read Owner-Write Parallel Random Access Machine *is the same as a SIMD CROW PRAM (see Def. 2.13) except for the following differences:*

- *Each processor has its own program counter, i.e., the processors share the same program, but they can execute different instructions at the same time. The processors are synchronized after every instruction. It means that all the processors start executing an instruction simultaneously and they can start the next instruction only after all of them finish the current instruction.*
- *The conditional assignment instruction* `if z>0 then x:=y` *and the conditional jump instruction* `if all_zero(z) goto label` *are missing.*
- *There is the RAM conditional jump instruction* `if x>=0 then goto label` *available.*

The CROW PRAM should be fully named the SPMD CROW PRAM for Single Program Multiple Data in our naming scheme, but we will omit the part "SPMD".

Theorem 2.19 *CROW PRAM* $\in \mathcal{C}_2$.

Proof.

Simulation of the SIMD CROW PRAM on the CROW PRAM:

> Lemma 2.14 gives $T^{\mathrm{PRAM}}_{\mathrm{SIMD}}(n) = \Omega(\log n)$ which bounds the number of processors by $p \leq O(2^{O(T^{\mathrm{PRAM}}_{\mathrm{SIMD}})})$ due to Lemma 2.15. Almost all the instructions can be simulated directly one by one. The only exception is the SIMD conditional jump instruction which involves a collective decision based on one local register in each processor — the `all_zero(z)` condition. Evaluation of this condition in all the processors simultaneously can be done using p binary trees of depth $\log p$, one tree rooted in each processor. Every tree is used to perform a conjunction of p values of registers z in time $O(\log^2 p) \leq O((T^{\mathrm{PRAM}}_{\mathrm{SIMD}})^2)$. Thus, the time complexity of the whole simulation is $T^{\mathrm{PRAM}}(n) = O((T^{\mathrm{PRAM}}_{\mathrm{SIMD}}(n))^3)$.

Simulation of the CROW PRAM on the SIMD CROW PRAM:

> Every SIMD processor simulates the corresponding PRAM processor. A PRAM processor has its own program counter which is stored in a local register of the SIMD processor. To simulate one step, the SIMD machine cycles through the whole PRAM program which has a constant size. In each processor, it simulates only the single instruction of the program determined by the value of the program counter of that processor. The simulation time is $T_{\text{SIMD}}^{\text{PRAM}}(n) \leq cT^{\text{PRAM}}(n)$ where $c > 0$ is a constant depending only on the number of instructions in the PRAM program.

□

We define other types of PRAMs. These variants differ from the CROW PRAM (or the SIMD CROW PRAM) only by the way how the global memory is accessed and how the read and write conflicts are resolved.

Definition 2.20 (PRAM memory access conflict resolution)

- ER (Exclusive Read) — *simultaneous reading by more than one processor from the same global register is forbidden,*
- CR (Concurrent Read) — *any number of processors can read the same global register in the same step,*
- OW (Owner Write) — *each global register has its owner, i.e., the only processor which is allowed to write into this register,*
- EW (Exclusive Write) — *simultaneous writing by several processors to the same global register is not allowed,*
- CW (Concurrent Write) — *in the same step, any number of processors may write into one global register; several conflict resolution protocols for the processors writing into the same register are possible:*
  - Common — *all processors are required to write the same value,*
  - Priority — *only the processor with the smallest index succeeds in writing, values written by other processors are lost,*
  - Arbitrary — *the resulting value of the register is unpredictably picked from all the written values,*
  - Combine — *a simple associative function computable in $O(\log^k n)$ time for some constant $k > 0$ and with two n-bit arguments (e.g., maximum, sum, OR, AND,...) is performed on the values being written and the result is stored in the register.*

A particular PRAM machine is then denoted by a combination of read and write access specification, e.g., Common CRCW PRAM.

Theorem 2.21 *Any ER, CR, OW, EW, and CW (Common, Priority, Arbitrary, or Combine) PRAM and SIMD PRAM is a member of the second machine class.*

Proof. We show that all these models are polynomially time-equivalent. We already know that CROW PRAM and SIMD CROW PRAM are in the second class (Theorems 2.17 and 2.19).

Simulation of any variant on a Combine CRCW PRAM:

An ER algorithm is a special case of a CR algorithm, similarly OW is a special case of EW, and OW and EW are special cases of CW. Common, Priority, and Arbitrary CRCW PRAMs can be trivially simulated by the Combine CRCW PRAM. The result of the combining function is equal to the value of its first argument.

CRCW can be simulated on EROW as follows:

Let us assume that the CRCW machine uses p processors and m global registers. The EROW computer will have $O(p + m + pm)$ processors and global registers. The first p processors (let us call them the computational processors) simulate the CRCW processors. Next there are m processors (called the memory processors), each holding the content of one CRCW global register in one of its local registers. The last $O(pm)$ processors form m binary trees used for routing values flowing between the CRCW processors and global memory, i.e., between the EROW computational processors and memory processors. Each tree has depth $\log p$ and p leaves.

Instructions which do not involve global memory are simulated directly in the same time. Instead of reading from global register g by processor i, a request is passed from the i-th computational processor to the i-th leaf of the g-th tree and up to root of the tree to memory processor g. Simultaneous read requests are combined in the inner nodes of the tree. The read value is passed the opposite way from memory processor g to all the computational processors which requested the value. A write operation is handled similarly. The value being written goes from the computational processor to the memory processor. The inner nodes of the trees do the conflict resolution according to the chosen protocol.

Using the upper bounds for the number of processors from Lemma 2.15 and for the largest address from Lemma 2.10, and the lower bound for the time complexity from Lemma 2.14, the time complexity of the simulation is $T_{\mathrm{EROW}}^{\mathrm{PRAM}} = O(T_{\mathrm{CRCW}}^{\mathrm{PRAM}} \log^2(pm)) \leq O((T_{\mathrm{CRCW}}^{\mathrm{PRAM}})^3)$.

□

2.4 Weak Parallel Models

The class of weak parallel machines $\mathcal{C}_{\mathrm{weak}}$ was defined in [79]. In that paper, the parallel Turing machine (PTM) is defined, analyzed, and its pipelined version (PPTM) is showed to be a member of $\mathcal{C}_{\mathrm{weak}}$. In this section, we briefly repeat important definitions and facts from [79] and present some new results: two restrictions of the PPTM model with proofs of their membership in the class of weak parallel machines and a theorem saying that the original nonrestricted PPTM is too strong to belong into $\mathcal{C}_{\mathrm{weak}}$. These results will be used later in Sections 3.3.4 and 4.5.4 when dealing with pipelined versions of BSP and dBSP.

Definition 2.22 (PTM) *A (non-pipelined) d-dimensional k-tape* Parallel Turing Machine *— (d, k)-PTM for short — is based on a nondeterministic sequential Turing machine (NTM)*

having k tapes of dimension d. The computation starts with only one processor (a set of k heads with the control unit realizing the transition relation is called the processor). If the NTM would do a nondeterministic choice of performing one of instructions $i_1, i_2, \ldots, i_b$, the PTM creates $b-1$ new processors and each of b processors executes a different instruction from the alternatives. Then all the processors work independently, but they share the same set of tapes. The processors run synchronously, performing one step per time unit. It is forbidden for several processors to write simultaneously different symbols into the same tape cell.

Whenever any processor encounters a choice in the transition relation, it generates new copies of itself. One would suspect that there could emerge $2^{O(T(n))}$ processors during $T(n)$ steps, but it is not the case. Any processors in the same state with heads at the same positions are indistinguishable and effectively act as a single processor. Assuming $S(n)$ cells occupied on each of k tapes and q possible states, at most $q(S(n))^k = O(S^k(n))$ distinguishable processors can exist. The limited number of processors causes that the PTM does not belong to the second machine class. As for membership in the first class, $(1,1)$-PTM $\in \mathcal{C}_1$. A multi-tape PTM and a sequential TM can simulate mutually each other, but the sequential TM can be simulated by a multi-tape PTM in sublinear space [79], thus (d,k)-PTM $\notin \mathcal{C}_1$ for $k > 1$.

Definition 2.23 (PPTM) *A* pipelined parallel Turing machine *(d,k)-PPTM has a two dimensional read-only input tape, and one dimensional write-only output tape. The i-th input word w_i is written from the beginning of the i-th row of the input tape. The machine prints its i-th output to the i-th cell of the output tape. The input is read in order $w_1, w_2, \ldots$ and the output is printed in the same order. The number of steps made by the PPTM between reading the first symbol of w_i and printing the i-th output depends only on the length of w_i. The PPTM halts after printing the last output.*

A PPTM $\mathcal{M}$ solves a decision problem[4] *$\mathcal{P}$ in* time $T(n)$ *iff for a sequence of instances of the same length n the machine $\mathcal{M}$ prints the i-th output after at most $T(n)$ steps after reading the first symbol of the i-th input.*

A PPTM $\mathcal{M}$ works in space $S(n)$ *iff any sequence of arbitrarily many inputs of the same length n is processed using at most $S(n)$ cells on working tapes.*

A PPTM $\mathcal{M}$ solves $\mathcal{P}$ with period $P(n)$ *iff it reads the first symbol of the input w_i after at most $P(n)$ steps after reading the first symbol of w_{i-1} and prints the i-th output at most $P(n)$ steps after printing the $(i-1)$-st output.*

Typically, the input words of a PPTM are *instances* of the same problem $\mathcal{P}$. Hence, the PPTM is especially suitable to solve not just one instance of $\mathcal{P}$, but a (long) sequence of instances.

Definition 2.24 *The PPTM is* uniform *iff it eventually starts cycling (repeating the same configuration) with period $P(n)$ on a sufficiently long sequence of identical inputs.*

The *Pipelined Computation Thesis* (PCT) says that for a problem $\mathcal{P}$, there is a pipelined algorithm for $\mathcal{P}$ with period polynomially equivalent to the sequential space complexity of $\mathcal{P}$. The class $\mathcal{C}_{\text{weak}}$ consists of exactly those machines which satisfy the PCT. The uniform $(1,1)$-PPTM satisfies half of the PCT, as is proved in the following lemma [79].

[4] equivalently: recognizes a language

Lemma 2.25 *A Turing machine computing in time $T(n)$ and space $S(n)$ can be simulated by a $(1,1)$-PPTM with period $P(n) = O(S(n))$, time $O(T(n))$ and space $O(T(n))$.*

Proof. Each instance, which is being computed, occupies $S(n)$ consecutive tape cells. There are $T(n)/S(n)$ such instances at any time, occupying $T(n)$ cells in total, thus the PPTM needs space $O(T(n))$. The pipelined machine simulates one step for each instance and moves the whole content of its working tape one cell to the right. This can be done in a constant number of steps with enough processors. After $S(n)$ steps, there is enough room for a new instance at the beginning of the working tape. At the same time, the oldest instance on the tape is finished and its output printed. Hence the period of computation is $O(S(n))$. After $T(n)/S(n)$ periods, $T(n)$ steps is performed on an instance and its processing is finished. The PPTM time complexity is $O(T(n))$, because each period comprises $S(n)$ steps. Arrangement of the instances on the working tape is drawn in Figure 2.1. □

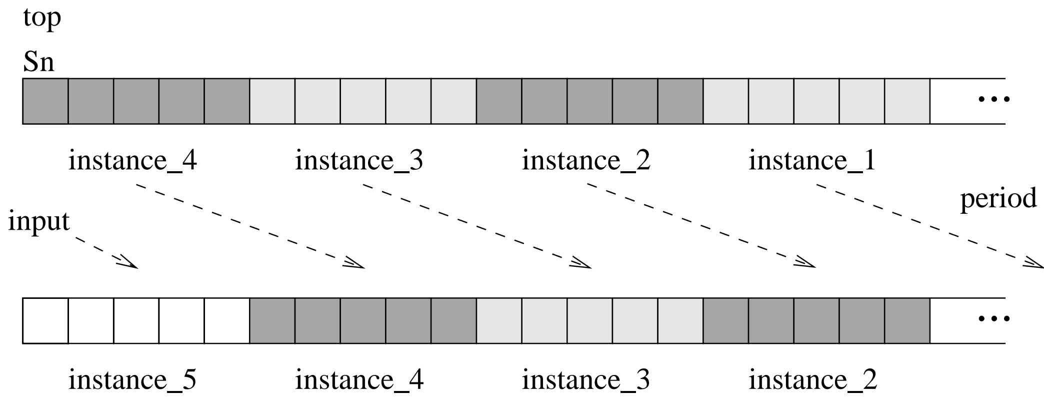

Figure 2.1: Pipelined PTM with period $P(n) = O(S(n))$

As we will see later, the second part of the PCT — simulation of PPTM period $P(n)$ in sequential space $O(P^k(n))$ — does not hold for general uniform PPTM. Nevertheless, a single period can be simulated in small space [79].

Lemma 2.26 *Let $\mathcal{M}$ be a uniform $(1,1)$-PPTM computing with period $P(n)$. Then there exists a sequential Turing machine $\mathcal{M}'$ with a separate input tape computing in space $S(n) = O(P(n))$, which gets a configuration of $\mathcal{M}$ on its input tape and checks that $\mathcal{M}$ returns into the same configuration after $P(n)$ steps.*

Proof. Simulation machine $\mathcal{M}'$ uses the fact that only processors with heads at cells $i - t, \dots, i+t$ can influence the content of cell i after t steps. Figure 2.2 shows which cells have to be remembered and which may be forgotten to be able to simulate $P(n)$ steps. First, symbols in cells $0, \dots, P(n)$ are copied from input to the working tape, content of the cell 0 after $P(n)$ steps is computed and checked, then the symbol in the cell $P(n) + 1$ is copied to the working tape and the cell 1 after $P(n)$ steps is evaluated, and so on until the whole working tape is processed. At any time, only a segment of $O(P(n))$ cells of the original PPTM $\mathcal{M}$ tape has to be stored on the working tape of $\mathcal{M}'$. Machine $\mathcal{M}'$ is a deterministic sequential Turing machine working in space $S(n) = O(P(n))$. See [79] for technical details of the simulation. □

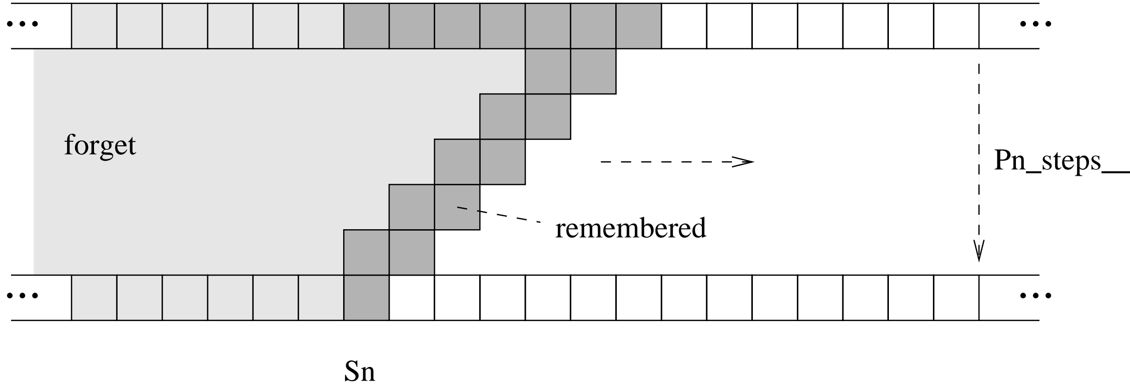

Figure 2.2: Sequential simulation of a pipelined PTM

The above algorithm allows to simulate a single period in polynomially related sequential space. This is not sufficient to claim that a whole computation can be simulated in polynomially related space, because the simulation does not cover the startup phase of the pipelined algorithm, before the cycle (required by uniformity) is entered. The cost of the startup phase of a uniform pipelined computation has no a priori upper bound and may contain a major part of the whole computation[5].

Now we present two restrictions of a uniform PPTM, namely the *restricted PPTM* and the *strictly pipelined PTM*. We prove that both models belong to the class of weak parallel machines.

The limited PPTM places an upper bound on the space complexity of the startup phase, i.e., the initial part of a computation before the machine enters a cycle required by uniformity. The desired space upper bound is $O(P^k(n))$ for some constant $k > 0$. This allows, together with Lemma 2.26, to space-efficiently sequentially simulate the whole PPTM computation.

Definition 2.27 (Limited PPTM) *A uniform (d,k)-PPTM with period $P(n)$ is called a limited (d,k)-PPTM iff the computation cycle*[6] *contains configuration C, such that there is a sequential Turing machine algorithm working in space $S(n) = O(P^k(n))$ for some constant $k > 0$, which gradually prints the cells of C from left to right*[7] *to its output tape, given the input of size n.*

Theorem 2.28 *Limited $(1,1)$-PPTM is a member of the class of weak parallel machines.*

Proof. A simulation of a space $S(n)$ sequential computation on a PPTM with period $P(n) = O(S(n))$ is provided by Lemma 2.25. The algorithm is limited, because the PPTM tape contains configurations of a space bounded sequential machine. Therefore parts of the tape corresponding to individual instances can be generated by the sequential algorithm in space $O(S(n))$.

The reverse simulation uses the algorithm from Lemma 2.26 to check that the PPTM returns into the same configuration after a period. The special configuration C (required to

[5]see Theorem 2.32
[6]which the computation is required to enter due to uniformity
[7]assuming the working tape starts in the left and stretches to the right

exist by the definition of the limited PPTM) is tested, because it can be generated in space $S(n) = O(P^k(n))$. If a new cell is needed by the cycle testing algorithm, it is obtained by running the C generating algorithm until one new cell is printed. □

Another possible approach is not to restrict the complexity of the startup phase, but instead to prevent usage of its results during subsequent computation. More generally, we forbid sharing of information among instances. Thus, all necessary data must be computed for each instance separately. The definition of a *strictly pipelined parallel Turing machine* formalizes the intuitive notion of the instances not interacting with each other. For each instance, there are tape cells which are "owned" by the instance. Only contents of these cells and processors with heads on them may have an influence on the computation of that instance.

Definition 2.29 *For a chosen input word w, a partitioning of tape cells of a $(1,1)$-PPTM into two sets is a partitioning into set $P_{w,t}$ of cells* pertinent *to w in step t and set $I_{w,t}$ of cells* independent *on w in step t, iff:*

1. $P_{w,t} \cap I_{w,t} = \emptyset$.
2. $P_{w,t} \cup I_{w,t}$ *contains the whole rewritten part of the working tape.*
3. *In the beginning of a computation ($t = 0$), there is only one processor, which is in the initial state and is scanning the cell number 0. At that moment, the cell $0 \in P_{w_1,0}$.*
4. *If the cell number $0 \in I_{w_i,t} \cap P_{w_i,t-1}$ then $0 \in P_{w_{i+1},t}$ and a new processor in the initial state scanning cell 0 is created.*
5. *A processor with the head on cell $c \in P_{w,t}$ in step t must not have its head on cell c' in step $t' > t$ if $c' \in I_{w,t'}$.*
6. *Only a processor with the head on cell $c \in P_{w_i,t}$ can read a symbol from the i-th input word w_i in step t.*
7. *Cell $c \in I_{w,t} \cap P_{w,t-1}$ contains the blank symbol λ in step t.*
8. *If a processor with its head on cell $c \in P_{w,t}$ creates new processor p then cell c' scanned by the head of p will belong to $P_{w,t+1}$. An exception is a processor created according to (4).*
9. *If a processor enters a terminal state, it disappears.*

Definition 2.30 (Strictly pipelined PTM) *A uniform $(1,1)$-PPTM is strictly pipelined, iff in the beginning of computation the working tape contains the blank symbol λ in every cell and the sets of cells pertinent to any pair of input words w_i and w_j, $i \neq j$ are disjoint at every step t.*

The definition of the strictly pipelined PTM ensures that computations of individual instances — corresponding to individual input words — are completely separated from each other. Due to (3), (4), and (5), every processor is bound to a particular instance during all time of the processor's existence. A processor pertinent to an instance can neither become independent, nor pertinent to another instance. The feature (4) provides a means for starting

computation of new instances. Conditions (7) and (8) guarantee that no instance can exploit information produced by another instance i, because information can be neither left in a cell which becomes independent on the instance i, nor passed outside the cells pertinent to i by a newly created processor. Passing information to the cells independent to i by an already existing processor is forbidden by (5). A processor pertinent to an instance i cannot look at an input word other than w_i, due to (6). Feature (9) prevents processors pertinent to finished instances from becoming obstacles in further computation. The cells pertinent to an input word are not marked in any way. The notion of pertinent and independent cells is just an abstract characteristic of the PTM. Thus, there could be many ways of partitioning the cells, all satisfying the conditions of Definition 2.29. A parallel Turing machine is strictly pipelined if and only if there exists at least one such partitioning.

Theorem 2.31 *Strictly pipelined* $(1,1)$*-PPTM is a member of the class of weak parallel machines.*

Proof. The strictly pipelined PTM is uniform, therefore it starts cycling after some time. During one cycle, a new input is read and computation of a new instance begins. Only a processor scanning cell 0 can start computation of a new instance. A new instance does not know how many instances have already started before, therefore it must enter the cycle. Otherwise the cycling behavior would not be guaranteed. During one period, i.e., $P(n)$ steps, the instance can allocate up to $O(P(n))$ tape cells. As cycling is required, these cells must be made free again in subsequent $O(P(n))$ steps for usage by the next instance. At the same time, only $O(P(n))$ new cells can be allocated. The instance cannot utilize more than $O(P(n))$ cells at any time and hence the computation of one instance can be simulated by a nonpipelined PTM in space $O(P(n))$. Since the nonpipelined $(1,1)$-PTM $\in \mathcal{C}_1$, space $S(n) = O(P^k(n))$ for some constant $k > 0$ suffices for a sequential Turing machine to simulate the computation.

The reverse simulation of a space $S(n)$ Turing machine on the pipelined PTM with period $P(n) = O(S(n))$, as described in Lemma 2.25, satisfies the restriction of the strictly pipelined PTM. □

We can ask a natural question whether the restrictions of limited and strictly pipelined PPTMs are necessary or not. We prove that general or even uniform PPTMs are too powerful for being weak parallel machines[8]. The problematic part is space bounded sequential simulation of a PPTM. Although we are able to simulate a single period (and thus all periods due to uniformity) in small space using Lemma 2.26, it is also necessary to obtain one configuration belonging to the cycle somehow. If we just nondeterministically guess a configuration and test that it will repeat after every $P(n)$ steps, we are not sure whether the PPTM would ever reach such a configuration. We could guess a perfectly cycling configuration unreachable from the initial configuration of the PPTM. Therefore, we must check that the guessed (or otherwise obtained) cycling configuration would be ever reached by the simulated PPTM machine. Generally, the PPTM can get into the cycle after an arbitrarily complex precomputation, which cannot be simulated using only $O(P(n))$ space. The precomputation phase is a major obstacle, as is shown in the following theorem and its corollaries.

Theorem 2.32 *Every sequential Turing machine (TM) algorithm that halts on each input can be simulated by a uniform* $(1,1)$*-PPTM algorithm with period* $P(n) = O(n)$.

[8] This fact disproves the claim of [79] that uniform PPTMs are members of $\mathcal{C}_{\text{weak}}$.

Proof. A PPTM computation is divided into two phases. In the precomputation phase, results of TM computations are computed for all inputs of length n. During the computation phase, a simple table lookup is performed for each input. The PPTM uses a two-track tape. The table is stored in one track and the inputs are put into the second track.

Precomputation phase:

> Run the sequential TM algorithm for each of $2^{O(n)}$ possible inputs and write a table of pairs $\langle i, o \rangle$ to the first track of the tape. For each input i occupying n tape cells there is a single cell o which holds the information about acceptance or rejection of this input. The table can be stored in space $n2^{O(n)} = 2^{O(n)}$. As the definition of the pipelined PTM requires that reading of a new input word is started after every $P(n)$ steps, the input is continuously read and stored on a separate track of the PTM's working tape.

Computation phase:

> Create one head for each tape cell to be able to move information along the tape in parallel. The leftmost head reads one new input symbol in every step. Simultaneously, the contents of the second track is shifted one cell to the right. After $O(n)$ steps, one input word is read and stored at the tape so that it is aligned with the first entry of the lookup table created in the precomputation phase. Now one head compares — sequentially, in time $O(n)$ — the input and part i of the first table entry. If it matches, the corresponding o is attached to the input. Then a new period begins. When a new input is read, the input from the first period is shifted to the right, aligned with the second entry of the table, and compared. After $2^{O(n)}$ periods the input gets to the end of the table. As the table contains all the possible inputs, exactly one match must have occurred by that time. Therefore, the result for the input is known and can be printed to the output. At the same time, $2^{O(n)}$ inputs can be simultaneously compared to different entries of the table. After every $O(n)$ steps, one new input is read and one output is produced. First, the inputs read and buffered on the working tape during the precomputation phase are processed. Simultaneously, the new inputs are read and put to the buffer in place of already processed ones.

□

Corollary 2.33 *There is a uniform $(1,1)$-PPTM computing with period $P(n)$ such that it cannot be simulated by a Turing machine in space $O(P^k(n))$ for some constant $k > 0$.*

Proof. Consider a problem with TM space complexity $S(n) = \omega(n^k)$ for all $k > 0$. It means that there is no TM which solves the problem in space less than $S(n)$. A TM algorithm for this problem can be simulated by a PPTM with period $P(n) = O(n)$ according to Theorem 2.32. A simulation of the PPTM on the TM in space $O(P^k(n)) = O(n^k)$ yields a contradiction to the assumption about $S(n)$. □

We showed that computations of a sequential algorithm on all inputs of fixed length can be performed in the precomputation phase and consequently a short period is achieved. Although it is possible to simulate one period in small sequential space, the complete simulation of the pipelined computation may need much more space. The mutual simulation between a sequential TM and a uniform $(1,1)$-PPTM with polynomially equivalent period and sequential space — which is required by the Parallel Computation Thesis — is possible from TM to PPTM, but not vice versa. Thus, we refuted the PCT for the uniform $(1,1)$-PPTM.

Corollary 2.34 *The uniform* $(1,1)$*-PPTM is not a model belonging to the class of weak parallel machines.*

Note that Corollary 2.34 is not in contradiction to Theorems 2.28 and 2.31, because the table lookup algorithm from Theorem 2.32 is neither limited nor strictly pipelined. The table of results needs space $2^{\Theta(n)}$ which exceeds the upper bound of a limited algorithm with polynomial period. Moreover, the table is reused by all instances which is forbidden by strict pipelining.

Chapter 3

Analysis of the BSP Model

In this chapter, we turn our attention to the Bulk Synchronous Parallel (BSP) model of parallel computation, introduced by Valiant in [71]. First we examine its original quite informal definition and point out its questionable details. We address the problems arising from the imprecise definition by giving another, formal definition of the BSP model. After that, the rest of the chapter is devoted to analysis of the BSP machines in the framework of machine classes $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_{\text{weak}}$.

3.1 Definition of BSP

In literature [49, 52, 71], the BSP model is described rather informally, in more or less intuitive terms. Such approach is usually sufficient for design and analysis of parallel algorithms, because practical algorithms are designed so as not to abuse some features of the model to reduce the computation complexity. But this is not our case. We are going to analyze efficiency of the BSP model in the context of machine classes and the massively parallel models from the second machine class. This task inevitably requires using of "dangerous" features, e.g., large word size and exponential number of processors. Therefore a carefully and precisely defined model is necessary.

We formalize the BSP model in this section. Not only a formal definition is given, but also the choice of individual features is discussed. There are many potential variants of the model different only in features which are often considered equivalent and therefore uninteresting. Nevertheless, for some of these features, a proper analysis shows that such an equivalence does not exist and we must be very careful to define the BSP model in such a way that it matches our intuition. We address this challenge by starting with a traditional informal "definition" (more correctly, a description) of the BSP model, based on its description in [52]. Then its — usually overlooked — problems, inaccuracy, and ambiguity are identified. By resolving all these problems, we finally come to a formal definition of the BSP which will suffice for our purposes. When developing the definition we also keep in mind the requirement of simple mutual simulations between the BSP and other models, especially the RAM and various PRAMs.

Definition 3.1 (BSP — an informal description)
The Bulk Synchronous Parallel Computer *consists of p processors with local memories. Every processor is essentially a RAM. The processors can communicate by sending messages via a*

router (some communication and synchronization device). The computation runs in supersteps, i.e., the processors work asynchronously, but are periodically synchronized by a barrier. A superstep consists of three phases: computation, communication, and synchronization. In the computation phase, the processors compute with locally held data. The communication phase consists of a realization of so-called h-relation, i.e., processors send point-to-point messages to other processors so that no processor sends nor receives more than h messages. The data sent in one superstep are available at their destinations from the beginning of the next superstep. In the final phase of each superstep, all the processors perform a barrier synchronization.

Performance of the router is given by two parameters g (the ratio of the time needed to send or receive one message to the time of one elementary computational operation — the inverse of the communication throughput) and l (the communication latency and the synchronization overhead). If a BSP computation consists of s supersteps and the i-th superstep is composed of w_i computational steps in every processor and of h_i-relation, then the time complexity of the computation is defined[1] *by*

$$T^{\mathrm{BSP}} = \sum_{i=1}^{s} (w_i + h_i g + l) = W + Hg + sl \ ,$$

where $W = \sum_{i=1}^{s} w_i$ and $H = \sum_{i=1}^{s} h_i$.

Now, we explain problems with the informal definition and ways how they can be solved. As it will become clear later, a solution of these technical details allows us to develop simple and efficient simulation algorithms of the BSP on the Turing machine and the RAM[2].

1. We must be more explicit about the BSP processors. As was explained earlier (in Sec. 2.2), logarithmic cost or a bounded word size is necessary for a processor to be a first class device. Otherwise any single BSP processor would already belong to the second class, which is undesirable. A BSP processor will be almost identical to the RAM from Def. 2.8 except for a special register containing processor's index and several additional communication and synchronization instructions. The new instructions are `send` (send a message to another processor), `recv` (receive a message), and `barrier` (notify other processors that this processor has nothing more to do in the current superstep, finish the communication, wait for all the processors to execute `barrier`, and begin a new superstep).

2. The communication is performed by sending point-to-point messages. Number h of sent or received messages by a processor in a superstep together with parameter g determine communication time hg. This is reasonable, if all the messages are approximately of the same length. To allow messages of any size and retain the simple communication time measure we define h to be the total number of bits in all the messages sent or received by the processor in a superstep. In the informal definition, it is not specified whether the message data are directly transferred by the router from the local memory of the source processor to the local memory of the destination processor, or whether some buffering of data is performed in the router. The exact moment when data are copied in and out of

[1] $\max\{w_i, h_i g, l\}$ is sometimes used instead addition, but these two definitions differ only by a constant factor not greater than 3.

[2] See, e.g., the proofs of Lemmas 3.8 and 3.11.

processors' memories should be specified and cost of this copying should be defined. We adopt an unbuffered-send buffered-receive scheme. When some interval of consecutive registers is marked by the `send` instruction to be sent as a message, the data are taken from these registers by the router only at the time of actual communication. The data transfer may occur any time unit after the `send` and before the end of the superstep. Therefore the data should not be altered till the next `barrier` instruction. No register can belong to more than one message sent in the same superstep. When communication is finished at the end of a superstep, messages are stored in buffers "between" the router and the destination processors. A destination processor can sequentially move messages — one at a time — from the buffer to its local memory by the `recv` instruction in any time during the superstep following the superstep when the messages were sent. In addition to hg cost, each `send` and `recv` is charged time according the size (in bits) of the address and the content of each register it involves.

3. Note that only the time complexity has been defined above. We are also interested in the space needed by BSP algorithms. It is crucial that not only the memory registers, but also the transient messages are included in the space complexity (or their total size is suitably limited). Otherwise, if only memory registers contributed to the space and we used fully buffered communication, i.e., `send` copied data to an internal buffer in the router, not counted to the space complexity, and the register containing the message could be immediately reused, then it would be possible to simulate any space S^{TM} Turing machine computation by the BSP in the space $S^{\mathrm{BSP}} = \log S^{\mathrm{TM}}$ as follows: TM tape cells are numbered. Cell i containing alphabet symbol s is represented by BSP message $\langle i, s\rangle$, which can be stored in two registers. The simulating BSP machine has two processors. Head position h is remembered in a register of the first processor. The simulation algorithm makes use of the fact that only the cell with number h can be altered in one step. One TM step is simulated by one BSP superstep. During a superstep, the first processor reads messages and remembers the single message $\langle h, s\rangle$ — the content of the cell with the head. All other messages are immediately sent back to the second processor. The message $\langle h, s\rangle$ is updated according to the TM transition function and the resulting $\langle h, s'\rangle$ is also sent to the second processor. The head position h is updated to $h' \in \{h-1, h, h+1\}$. The second processor just sends every message it receives back to the first processor. The only data of nonconstant number of bits are the cell numbers which are bounded from above by $\log S^{\mathrm{TM}}$. At any time, all the messages except one in each processor are stored (for free) in the router.

4. We assume that a BSP machine has as many processors as needed by the algorithm at hand. The number of processors is a nondecreasing function of the input size, i.e., $p = p(n)$. In the beginning of a computation, a finite number of processors is active. additional processors can be activated by sending messages to them. It is reasonable to expect that communication and synchronization become increasingly complicated as the number of processors grows. Therefore in general, communication parameters g and l are not constants, but they become nondecreasing functions dependent on the number of processors: $g = g(p)$, $l = l(p)$.

The formal BSP definition is quite long, therefore it is split into three parts. Definition 3.2 describes the structure, the instruction set, and computation of a BSP computer. The following Def. 3.3 and 3.4 establish the time and space complexities of BSP computations.

Definition 3.2 (BSP) *A* Bulk Synchronous Parallel Computer *with $p = p(n)$ processors and communication parameters g and l is denoted* $\mathrm{BSP}(p, g, l)$. *Parameters $g = g(p) \geq 1$ (inverse of the communication bandwidth*[3]*) and $l = l(p) \geq 1$ (communication latency and synchronization cost) are nondecreasing functions of p. Every processor is a RAM as defined in Def. 2.8 with following extensions:*

- *Each processor has read-only register* `pid` *which contains the index of the processor. The processors are indexed $0, 1, \ldots, p-1$.*
- *The instruction* `send(addr,len,pid)` *sends a message to the processor with index* `pid`*. The message content is read from registers* `addr`, `addr`$+1, \ldots,$ `addr`$+$ `len`-1.
- *The instruction* `recv(addr,len,src)` *receives a message of length* `len` *from processor* `src` *into registers* `addr`, `addr+1`$, \ldots,$ `addr`$+$ `len`-1. *Only* `addr` *is specified by the program,* `len` *and* `src` *are output parameters of the instruction. If there are no more messages to be received, -1 is stored in* `src` *and* `len` *is set to 0.*
- *The instruction* `barrier` *terminates a superstep.*

The processors share the same program, but they compute independently — unlike in PRAM (see Def. 2.18), they are not synchronized after each instruction. The computation is divided into supersteps*. The instruction* `barrier` *is a mechanism for periodic barrier synchronization of the processors. Each processor terminates the current superstep by calling* `barrier` *and waits until all the processors call* `barrier`*. Only then the next superstep begins.*

The processors communicate by sending and receiving point-to-point messages via a router using instructions `send` *and* `recv`*. After a* `send` *instruction sends the content of register*[4] *r, a valid BSP algorithm must ensure that register r will not be changed till the end of the current superstep and that it will not be sent by any subsequent* `send` *instruction during the current superstep. Register r can be used freely again after the next* `barrier` *instruction. All messages sent in a superstep are ready to be received by their destination processors at the beginning of the next superstep. Till the end of the current superstep, each processor must receive all messages sent to it during the previous superstep. Let us denote h_i^{send} the total number of bits of register contents sent by processor i during a superstep. Similarly, h_i^{recv} is the total number of bits received during the superstep. Let us define $h = \max_{0 \leq i < p}\{h_i^{\mathrm{send}}, h_i^{\mathrm{recv}}\}$. Then we say that the communication in the superstep has a form of* h-relation.

Input x of a computation is distributed in local memories of the first $O(|x|)$ processors in a way as assumed by the algorithm. The whole input must be read during the computation. In the beginning of a computation on input x, a certain number (defined by the algorithm, but at most $O(|x|^k)$ for some constant $k > 0$) of processors are active. Other processors are inactive, i.e., they are idle until they are activated. An originally inactive processor can be activated by sending a message to it. The newly activated processor begins its computation in the next superstep. The algorithm defines how the output of the computation is distributed among processors active at the end of the computation.

[3] Communication bandwidth is the number of bits which can be transferred to/from a processor in unit time.

[4] i.e., `addr` $\leq r \leq$ `addr` $+$ `len` -1

Definition 3.3 (BSP time complexity) *Assume a computation of* $\mathrm{BSP}(p, g(p), l(p))$ *having s supersteps. The time cost of the BSP instructions is the same as for the RAM instructions, i.e., an instruction with the largest operand x takes $|x|$ time units, where $|x|$ is the number of bits of x. The additional instructions have costs:*

$$\begin{aligned}
\texttt{send(addr,len,pid)}: \quad T^{\mathrm{send}} &= \sum_{i=0}^{\mathtt{len}-1} \Big(\log(\mathtt{addr}+i) + |r_{\mathtt{addr}+i}|\Big) + \log \mathtt{pid} \,, \\
\texttt{recv(addr,len,src)}: \quad T^{\mathrm{recv}} &= \sum_{i=0}^{\mathtt{len}-1} \Big(\log(\mathtt{addr}+i) + |m_i|\Big) + \log \mathtt{src} \,, \\
\texttt{barrier}: \quad T^{\mathrm{barrier}} &= 1 \,,
\end{aligned}$$

where r_i is the content of the i-th register and m_i is the i-th value in the message.

Let $w_{i,j}$ be the sum of the costs of all instructions performed by the j-th processor in the i-th superstep and $w_i = \max_{0 \le j < p}\{w_{i,j}\}$. Let the communication pattern of the i-th superstep be a h_i-relation. Then we define the time complexity *of the BSP computation as*

$$T^{\mathrm{BSP}} = \sum_{i=1}^{s} \Big(w_i + h_i g(p) + l(p)\Big) = W + Hg(p) + sl(p) \,,$$

where $W = \sum_{i=1}^{s} w_i$, $H = \sum_{i=1}^{s} h_i$, and p is the total number of processors used during the computation[5]*.*

Definition 3.4 (BSP space complexity) *The space complexity of a* $\mathrm{BSP}(p, g, l)$ *computation is*

$$S^{\mathrm{BSP}} = \sum_{i=0}^{p-1} \sum_{j} \Big(|j| + |l_{i,j}|\Big) \,,$$

where j goes through all used local registers in processor i, $|j|$ is the number of bits in the address of register j, and $|l_{i,j}|$ is the maximum number of bits stored during the computation in the j-th local register of processor i. Register **pid** *contributes to the space complexity too, but input registers which are not changed during computation and output registers which are only written but not read are not counted to the space complexity.*

Buffers for storing messages in the router are not part of the space complexity. But the requirement that each sent register is sent only once and not modified afterwards during a superstep ensures that the amount of data stored in internal buffers of the router is never larger than S^{BSP}. Therefore adding the size of transient messages to the space complexity could only double S^{BSP}. The total size of transient messages is limited by the space complexity, hence hiding a major part of the space complexity in messages, as described on page 31, is not possible.

3.2 Elementary Features of the BSP Model

We have arrived at a formal definition of the BSP computational model that is free from deficiencies of the informal model. Before we start examining how it is related to other

[5] not only the active processors in a particular superstep

sequential and parallel machine models, in this section we present some technical lemmas concerning BSP. Then we develop a simple simulation of a BSP computer on a RAM and derive its time and space overheads. This simulation will be extensively utilized later, in the following section.

Lemma 3.5 *The maximum number of processors used in the first T time units of a BSP computation on an input of size n is bounded from above by*

$$p_{\max} \leq 2^{O(\log n + T)} .$$

Proof. The idea is similar to Lemma 2.15. A new processor i is activated using instruction `send(addr,len,`i`)` instead of `activate(`i`)`, but it is still valid that the activating instruction takes $\Omega(\log i)$ time units (even without taking communication and synchronization cost into account). As this instruction must be finished in T time units, it places an upper limit to the value of i: $i \leq 2^{O(T)}$. □

Lemma 3.6 *Time and space complexities of a BSP computer are related by the inequality*

$$S(n) \leq pW(n) \leq pT(n) .$$

If $T(n) = \Omega(\log n)$ then

$$S(n) \leq 2^{O(T(n))} .$$

Proof. A processor of a BSP computer cannot allocate more than $T(n)$ bits in its local registers during $T(n)$ time units. Allocation of registers can occur only in the computational phase of a superstep, thus only $W(n)$ time units contribute to the bound. The sum over all p processors yields the result. See also the proofs of Lemmas 2.11 and 2.16.

Combination of the first part of this lemma with Lemma 3.5 yields $S(n) \leq pT(n) \leq 2^{O(T(n))}.T(n) \leq 2^{O(T(n)+\log T(n))} \leq 2^{O(T(n))}$. □

Lemma 3.7 *A BSP computer can simulate a deterministic Turing machine simultaneously with a polynomial slow-down and a linear space overhead.*

Proof. The simulation runs on a BSP with a single processor. Such a BSP computer is essentially equivalent to a RAM which is in the first machine class according to Theorem 2.12. □

Lemma 3.8 *A* BSP *computer with p processors and arbitrary g and l can be simulated by a RAM in time $T^{\mathrm{RAM}} = O(pT^{\mathrm{BSP}} \log p)$ and space $S^{\mathrm{RAM}} = O(S^{\mathrm{BSP}} \log p)$.*

Proof. Let us have a BSP(p, g, l) machine working in time T^{BSP} and space S^{BSP}. The contents of local BSP registers are stored in the RAM memory so that BSP register i of processor j corresponds to RAM register $3(ip + j)$. Additional space is used as buffers for holding the sent messages until they are received by their destination processors. According to Def. 3.2, each BSP register can belong to only one message sent during a single superstep. Thus the maximum number of buffer registers is $O(S^{\mathrm{BSP}})$. BSP register i of processor j has its associated RAM buffer register $3(ip + j) + 1$. The registers with indices of the form $3(ip + j) + 2$ are reserved for maintaining linked list of messages with the same destination processors. Records containing the address of the first register and the length of each message

sent to j-th processor are stored in the j-th list. Maintaining the message buffers and the linked lists is covered by time needed for execution of `send` and `recv` instructions multiplied by some small constant factor. Storing the individual local memories of the BSP processors in single large RAM memory inevitably put some registers to higher addresses. A BSP address can grow by up to $O(\log p)$ bits. Consequently, the simulation can be performed in space $S^{\mathrm{RAM}} = O(S^{\mathrm{BSP}} \log p)$. The RAM sequentially simulates the supersteps of the BSP computation.

program BSPonRAM;
procedure superstep;
    for $j := 0$ **to** $p-1$ **do begin**
        simulate computation of processor j during the superstep
            (read received messages from the buffer registers);
        move message data from the regular to the buffer registers;
    end
end;
begin
    while at least one processor has not halted yet **do**
        superstep
end.

Program `BSPonRAM` runs in $O(pT^{\mathrm{BSP}})$ computational steps, but larger addresses can slow down the simulation to $T^{\mathrm{RAM}} = O(pT^{\mathrm{BSP}} \log p)$. □

The factor "$\log p$" in the space complexity of the algorithm from Lemma 3.8 is an upper bound, which seems to be too high. The $\log p$ simulation overhead is caused by adding $\log p$ bits to each address. But this would enlarge space $(\log p)$-times only if BSP space complexity per processor was a constant. Intuitively, each processor contributes at least $\log p$ bits to the space complexity due to its *pid* register. Thus any additional space overhead induced by the simulation algorithm should be hidden by these $\log p$ bits and the space complexity would be only $S^{\mathrm{RAM}} = O(S^{\mathrm{BSP}})$. The following lemma shows that this reasoning is not true.

Lemma 3.9 *The simulation algorithm from Lemma 3.8 needs, in the worst case, space at least* $S^{\mathrm{RAM}} = \Omega(S^{\mathrm{BSP}} \log p / \log\log p)$.

Proof. Let us have a BSP algorithm running on p processors and using first r local registers of each processor. In each processor, a constant number of registers (including register `pid`) can contain $\log p$ bit values, the rest have values of up to $\log r$ bits. It yields BSP space $S^{\mathrm{BSP}} = O(p \log p + pr \log r)$. The simulating RAM needs space $S^{\mathrm{RAM}} = \Omega\Big(p \log p + pr(\log r + \log p)\Big)$. If, e.g., $r = \log p$ then $S^{\mathrm{BSP}} = O(p \log p \log\log p)$ and $S^{\mathrm{RAM}} = \Omega(p \log^2 p)$. We obtained a superlinear space overhead $S^{\mathrm{RAM}}/S^{\mathrm{BSP}} = \Omega(\log p / \log\log p)$. □

Now we have a lower bound on space $S^{\mathrm{RAM}} = \Omega(S^{\mathrm{BSP}} \log p / \log\log p)$, which does not match the upper bound $S^{\mathrm{RAM}} = O(S^{\mathrm{BSP}} \log p)$ from Lemma 3.8, but is significantly greater than $O(S^{\mathrm{BSP}})$. Time bound $T^{\mathrm{RAM}} = O(pT^{\mathrm{BSP}} \log p)$ is tight, because it is reached when simulating any BSP algorithm which operates only with values containing a constant number of bits. Nevertheless, for practical BSP algorithms, the RAM time is typically less than the upper bound, but can be greater than $O(pT^{\mathrm{BSP}})$.

3.3 BSP and Machine Classes

The BSP model is parametrized by the values of number of processors p, communication cost g and synchronization cost l. In this section, we will study how choice of these three parameters influences the computational power of BSP computers. We show that with large enough values of parameters g and l a BSP machine belongs to the first class and hence is in this sense equivalent to the standard sequential models. On the contrary, if there are sufficiently many processors and parameters g and l have small values, a BSP is a member of the second class, being as powerful as, e.g., a PRAM. We also show that there is a "shadow zone" for the values of the BSP parameters which causes the BSP computer to fall "somewhere in between" the first and the second class. Although we could suspect that such a machine belongs to the class of weak parallel machines, we prove the opposite.

3.3.1 BSP Computers in the First Machine Class

We begin with a simple case of a BSP machine with a fixed number of processors. It can be easily shown that such a BSP belongs to the first class. Hence, if we want a more powerful model, the number of processors must grow with the input size.

Theorem 3.10 *Let p be a constant (independent on the input size). Then a* $\mathrm{BSP}(p, g, l)$ *computer with arbitrary values of parameters g and l is a member of the first machine class.*

Proof. A Turing machine can be simulated on the BSP according to Lemma 3.7 with a polynomial time overhead and a linear space overhead. The reverse simulation follows from Lemma 3.8. Constant number of processors p yields $T^{\mathrm{RAM}} = O(T^{\mathrm{BSP}})$ and $S^{\mathrm{RAM}} = O(S^{\mathrm{BSP}})$. □

The superlinear space overhead lower bound proved in Lemma 3.9 for the simulation algorithm from Lemma 3.8 means that we cannot use this algorithm to establish membership in the first class for a BSP computer with a nonconstant — i.e., dependent on the input size — number of processors. Hence a different sequential simulation of the BSP model is needed. We develop an alternative simulation of the BSP by the Turing machine, which achieves a linear space overhead at cost of a larger execution time.

Lemma 3.11 *Any* $\mathrm{BSP}(p, g, l)$ *computer can be simulated by a deterministic Turing machine in time $T^{\mathrm{TM}} = O(p^3 (T^{\mathrm{BSP}})^3)$ and space $S^{\mathrm{TM}} = O(S^{\mathrm{BSP}})$.*

Proof. The local memory of each BSP processor is stored in a continuous region of the Turing machine tape in tuples $\langle i, r_i \rangle$, where i is the index of a used register and r_i is the value stored in the register[6]. All numbers are written in binary form. The beginning of the local memory of the next processor is marked by a special symbol. The space occupied on the tape is clearly bounded by $O(S^{\mathrm{BSP}})$. The second TM tape is used to store sent messages until they are received during the next superstep. As during a single superstep each register can be a part of only one message (see Def. 3.2), the space needed on the second tape is also bounded by $O(S^{\mathrm{BSP}})$.

The simulation algorithm is essentially the same as in Lemma 3.8. In the worst case, each of $O(pT^{\mathrm{BSP}})$ steps of individual BSP processors requires traversing the whole tape, taking

[6]See also the proof of Theorem 2.12

$O(S^{\mathrm{BSP}})$ time units. If a multiplication or division instruction, taking $|r_i|$ time units on the BSP, is simulated in time $|r_i|^2$ on the TM, an additional factor of $O(S^{\mathrm{BSP}})$ is introduced. The total simulation time is then $T^{\mathrm{TM}} = O(pT^{\mathrm{BSP}}(S^{\mathrm{BSP}})^2)$. Using Lemma 3.6 we obtain $T^{\mathrm{TM}} = O(p^3(T^{\mathrm{BSP}})^3)$. □

The upper time bound of the algorithms which simulate a BSP computer on a RAM or a Turing machine can be further improved by a careful analysis. For the moment, we do not consider the total T^{BSP} time, but we distinguish its computation, communication, and synchronization parts, according to Def. 3.3: $T^{\mathrm{BSP}} = W + Hg + sl$. The crucial observation is that the Turing machine time depends only on the computation, i.e., W, component of the BSP time.

Lemma 3.12 *Let a* BSP(p, g, l) *machine run in time* $T^{\mathrm{BSP}} = W + Hg + sl$. *Then this BSP computer can be simulated:*

- *by a RAM in time* $T^{\mathrm{RAM}} = O(pW \log p)$ *and space* $S^{\mathrm{RAM}} = O(S^{\mathrm{BSP}} \log p)$,
- *by a Turing machine in time* $T^{\mathrm{TM}} = O(p^3W^3)$ *and space* $S^{\mathrm{TM}} = O(S^{\mathrm{BSP}})$.

Proof. Manipulation with messages by the RAM and the Turing machine algorithms is completely covered by time of simulation of the `send` and `recv` instruction. Messages are only copied between RAM registers (or Turing machine tape cells, respectively), thus the Hg communication time is eliminated. The simulation algorithm first performs the computation done by the first BSP processor during one complete superstep, then the computation of the second BSP processor, and so on until the current superstep is finished by all p BSP processors. Only then the simulation of the next superstep starts. Hence we get the synchronization for free, eliminating term sl. □

The simulation method of Lemma 3.11 and its improvement by Lemma 3.12 will serve as a tool to prove membership of BSP machines in the first machine class for suitable values of their communication parameters g and l. The next two theorems show that, despite arbitrarily large number of available processors, if at least one of two functions $g(p)$ and $l(p)$ is at least polynomial in p — or even any root of p — then the BSP$(p, g(p), l(p))$ computer is a member of the first machine class. Note that both theorems hold also for g and l of growing faster than the stated bound. If both g and l grow too fast, it is always possible to run the sequential algorithm using only one processor, thus completely avoiding communication. In such case, we get the required mutual simulation with some first class machine from Theorem 3.10.

Theorem 3.13 *Let* $l(p) = \Omega(p^a)$ *for some constant* $a > 0$. *Then for any* p *and* $g(p)$, *the machine* BSP$(p, g(p), l(p))$ *is a member of the first machine class.*

Proof. Lemma 3.7 immediately yields the required simulation of the Turing machine on the BSP computer.

The reverse simulation of a BSP computer on a Turing machine runs according to the algorithm from Lemma 3.11. We immediately get the linear space overhead from the lemma. The only part left is to evaluate the time overhead. From Lemma 3.12 we know that $T^{\mathrm{TM}} \leq cp^3W^3$ for some constant $c > 0$. The assumption of the theorem yields $T^{\mathrm{BSP}} \geq W + skp^a$ for some positive constants k and a. Note that number of supersteps s is always at least one. We distinguish two cases:

1. $p \geq W \Rightarrow T^{\mathrm{TM}} \leq cp^6 = ck^{-6/a}(kp^a)^{6/a} \leq ck^{-6/a}(T^{\mathrm{BSP}})^{6/a}$,
2. $p \leq W \Rightarrow T^{\mathrm{TM}} \leq cW^6 \leq c(T^{\mathrm{BSP}})^6$.

In both cases, $T^{\mathrm{TM}} = O((T^{\mathrm{BSP}})^{\max\{6,6/a\}})$ holds. □

Theorem 3.14 *Let $g(p) = \Omega(p^a)$ for some constant $a > 0$. Further let at least one message be sent during the BSP computation. Then for any p and $l(p)$, the machine* BSP$(p, g(p), l(p))$ *is a member of the first machine class.*

Proof. The only difference in comparison with the proof of Theorem 3.13 is the time bound of the simulation of the BSP computer on the Turing machine. Again, we have $T^{\mathrm{TM}} \leq cp^3W^3$ for some constant $c > 0$, following from Lemma 3.12. Here we use another estimation of the BSP time: $T^{\mathrm{BSP}} \geq W + Hkp^a$ for some constants $k > 0$ and $a > 0$. The additional assumption of the theorem ensures that $H \geq 1$. The analysis of the two possible cases for the relation between p and W is exactly the same as in the previous proof. □

3.3.2 BSP in the Second Machine Class

In the previous section, we have seen that a BSP machine with a limited number of processors or with an arbitrary number of processors but with polynomial lower bound on $g(p)$ or $l(p)$, i.e., with slow communication, belongs to the first class. Now we will study BSP computers having many processors with ability of fast communication. First we prove that a BSP with constant values of $g(p)$ and $l(p)$ is a member of the second class. This result will be then extended to polylogarithmic communication parameters.

Lemma 3.15 *Let values of* BSP$(p, g(p), l(p))$ *parameters be $g(p) = O(1)$ and $l(p) = O(1)$. Let number of processors p be potentially unlimited, i.e., as many processors as needed can be used during a computation. Then such a BSP computer is a member of the second machine class.*

Proof.

Simulation of a BSP by an EREW PRAM:

The first p PRAM processors simulate one-to-one the p BSP processors. The additional p^2 PRAM processors are reserved for communication. Processor $(i+1)p+j$ maintains a list of messages sent from processor i to processor j during a superstep.

1. Local computation of the BSP processors is directly simulated by the corresponding PRAM processors.
2. Communication:
   (a) Each message sent by processor i to processor j is transferred to processor $(i+1)p+j$ using global register $(i+1)p+j$ for communication.
   (b) At the end of every superstep, 0 or 1 is put to global register $ip+j$, indicating whether there is at least one message $i \to j$.
   (c) For each j a sublist of ones is extracted from the list $0p+j, 1p+j, \ldots, (p-1)p+j$. This is done in parallel by p teams of p processors using the pointer jumping technique. The j-th list corresponds to the indices of processors which have sent a message to processor j during the superstep.

(d) The list is used in the next superstep by processor j to obtain messages sent to it. The messages are read from processors $(i+1)p+j$ — again using global register $(i+1)p+j$ for communication — but only for those i's which are in the list.

3. Synchronization: After simulating an instruction, processor i stores 1 in the i-th global register, if and only if the i-th BSP processor has finished the current superstep. Otherwise, processor i stores 0 in global register i. Using a binary tree for these values, it is determined whether all processors have written 1 and are ready to start the new superstep. If not, the processors that have already finished wait until the busy ones also finish their work. Then the new superstep begins.

The BSP computation time divided into its computational, communication, and synchronization parts is $T^{\mathrm{BSP}} = W + Hg + sl$. The total time of step (1) over all supersteps is $T_1^{\mathrm{PRAM}} = O(W)$. The cost of the `send` and `recv` instructions and the fact that each register can be a part of only one message during one superstep[7] ensure that $H \leq W$ and that steps (2a), (2b), and (2d) take up to $T_{2a}^{\mathrm{PRAM}} = T_{2b}^{\mathrm{PRAM}} = T_{2d}^{\mathrm{PRAM}} = O(W)$ during the whole computation. Steps (2c) and (3) take $O(\log p)$ computational steps each and thus $O(\log^2 p)$ time per superstep. Over all the supersteps, $T_{2c}^{\mathrm{PRAM}} = T_3^{\mathrm{PRAM}} = O(s \log^2 p)$. The total execution time of the PRAM algorithm is then $T^{\mathrm{PRAM}} = T_1^{\mathrm{PRAM}} + T_{2a}^{\mathrm{PRAM}} + T_{2b}^{\mathrm{PRAM}} + T_{2c}^{\mathrm{PRAM}} + T_{2d}^{\mathrm{PRAM}} + T_3^{\mathrm{PRAM}} = O(W + s\log^2 p)$. Clearly $W \leq T^{\mathrm{BSP}}$ and $s \leq T^{\mathrm{BSP}}$. According to Lemma 3.5, $\log p = O(T^{\mathrm{BSP}})$, assuming $T^{\mathrm{BSP}}(n) = \Omega(\log n)$. Hence, $T^{\mathrm{PRAM}} = O((T^{\mathrm{BSP}})^3)$.

Simulation of an EREW PRAM by a BSP:

Let us assume an EREW PRAM which uses p processors and r global registers during its computation. We know from Lemma 2.15 that $p \leq 2^{O(T^{\mathrm{PRAM}})}$ if $T^{\mathrm{PRAM}}(n) = \Omega(\log n)$. The simulating BSP machine has $p+r$ processors. The first p of them simulate the corresponding PRAM processors, the remaining r processors simulate the global registers (one processor per register). One PRAM step is simulated by one BSP superstep. Reading and writing of global memory registers is simulated by communication with the additional processors. The EREW property guarantees that all h-relations are 1-relations. An access to global register g (having $\log g$ bits long index) is simulated by communication with processor $p+g$ (with index having $\log(p+g) \leq \log p + \log g \leq O(T^{\mathrm{PRAM}}) + \log g$ bits). Therefore the computation is slowed by a factor up to $O(T^{\mathrm{PRAM}})$, yielding $T^{\mathrm{BSP}} = O((T^{\mathrm{PRAM}})^2)$.

□

Theorem 3.16 *Let $g(p) = O(\log^a p)$ and $l(p) = O(\log^b p)$ for some constants $a > 0$ and $b > 0$. Then the* BSP$(p, g(p), l(p))$ *computer with potentially unlimited number of processors is a member of the second machine class.*

Proof. BSP and PRAM are mutually simulated according to Lemma 3.15. Larger values of g and l can only make BSP slower, thus time of simulation of the BSP by the PRAM is not longer than in the lemma. The reverse simulation is slower by a factor $O(g(p) + l(p)) = O(\log^{\max\{a,b\}} p) = O((T^{\mathrm{PRAM}})^{\max\{a,b\}})$. Thus the BSP time is $T^{\mathrm{BSP}} = O((T^{\mathrm{PRAM}})^{2+\max\{a,b\}})$.

□

[7] See Def. 3.2 and 3.3.

An interesting practical observation is the fact that the BSP model implemented by a d-dimensional mesh network has parameter values $g(p) = l(p) = p^{1/d}$. Hence it is in the first machine class. A mesh with $d \leq 3$ is a scalable architecture in the 3-dimensional space. On the other hand, BSP implemented by a hypercube machine has $g(p) = l(p) = \log p$ and thus it is a member of the second class. Unfortunately, the hypercube is not a scalable interconnection network topology [75, 80].

3.3.3 Outside the First and the Second Machine Classes

The previous two sections gave a precise characterization of BSP computers with unlimited number of processors belonging to the first or the second machine class. A machine having values of parameters not yet analyzed, i.e., both $g(p)$ and $l(p)$ greater than polylogarithmic and less that any root of p, does belong neither to the first nor the second class. This result is formulated in Theorem 3.19. Before stating the theorem, we prove two auxiliary lemmas regarding mutual simulation of RAMs and BSPs.

Lemma 3.17 *Let there be a RAM algorithm with space complexity $S^{\mathrm{RAM}}(n)$ for a problem $\mathcal{P}$. Then there is a* BSP$(p, 1, 1)$ *algorithm running in time $T^{\mathrm{BSP}}(n) \leq O((S^{\mathrm{RAM}}(n))^6)$ on $p \leq 2^{O(S^{\mathrm{RAM}}(n))}$ processors for problem $\mathcal{P}$.*

Proof. The transitive closure algorithm from the proof of Theorem 2.17 (the second part, simulation of a TM by a PRAM) can be run on an EREW PRAM in the same time and with the same number of processors, i.e., $T^{\mathrm{PRAM}}(n) = O((S^{\mathrm{RAM}}(n))^3)$ and $p = 2^{O(S^{\mathrm{RAM}}(n))}$ processors, using $r = 2^{O(S^{\mathrm{RAM}}(n))}$ global registers. The PRAM is simulated on the BSP using the algorithm from the proof of Lemma 3.15 in time $T^{\mathrm{BSP}}(n) = O((T^{\mathrm{PRAM}}(n))^2)$ with $p + r = 2^{O(S^{\mathrm{RAM}}(n))}$ processors. □

Lemma 3.18 *Let there be an optimal RAM algorithm with time complexity $T^{\mathrm{RAM}}_{\mathrm{opt}}(n)$ for problem $\mathcal{P}$. Let $T^{\mathrm{BSP}} = W + Hg + sl$ be the time complexity of a BSP algorithm for $\mathcal{P}$. Then the computational part W and the number of processors p are related according to the formula*

$$\exists c > 0\, \forall \varepsilon > 0 : \; p \geq \left(c \frac{T^{\mathrm{RAM}}_{\mathrm{opt}}}{W} \right)^{\frac{1}{1+\varepsilon}} \geq \left(c \frac{T^{\mathrm{RAM}}_{\mathrm{opt}}}{T^{\mathrm{BSP}}} \right)^{\frac{1}{1+\varepsilon}} .$$

Proof. We exploit Lemma 3.12. From the optimality of the RAM algorithm follows $T^{\mathrm{RAM}}_{\mathrm{opt}} \leq kWp \log p$ for some constant $k > 0$, and thus $p \log p \geq 1/k \frac{T^{\mathrm{RAM}}_{\mathrm{opt}}}{W}$. The relation $\forall \varepsilon > 0 : \log p \leq o(p^{\varepsilon})$ yields $\exists c = 1/k > 0\, \forall \varepsilon > 0 : p^{1+\varepsilon} \geq c \frac{T^{\mathrm{RAM}}_{\mathrm{opt}}}{W}$. This gives the first inequality in the lemma. The second inequality holds because $T^{\mathrm{BSP}} \geq W$. □

Theorem 3.19 *Assume there exists*[8] *a problem $\mathcal{P}$ such that the fastest RAM algorithm for $\mathcal{P}$ has time complexity $T^{\mathrm{RAM}}(n) \geq c^n$ for some constant $c > 0$ and space complexity $S^{\mathrm{RAM}}(n) \leq c'n$ for a constant $c' > 0$. Let $\forall a, k > 0 : \log^k p < g(p) < p^a \wedge \log^k p < l(p) < p^a$. Then the* BSP$(p, g(p), l(p))$ *computer with potentially unlimited number of processors is neither a member of the first machine class nor the second class.*

[8] Existence of such a problem depends on the relation between classes P and $PSPACE$.

Proof.

$\mathrm{BSP}(p, g(p), l(p)) \notin \mathcal{C}_1$:

According to Lemma 3.17, there is an algorithm running on $\mathcal{M} = \mathrm{BSP}(p,1,1)$, such that $\exists b > 0 : p = b^{S^{\mathrm{RAM}}(n)}$ and $\exists d > 0 : T^{\mathcal{M}}(n) \le d.(S^{\mathrm{RAM}}(n))^6$. When run on a $\mathrm{BSP}(p, g(p), l(p))$ computer, the same algorithm is slowed down by factor $m(p) = \max\{g(p), l(p)\}$, i.e., $\exists b > 0\, \exists d > 0\, \forall a > 0 : T^{\mathrm{BSP}}(n) \le d \cdot (S^{\mathrm{RAM}}(n))^6 m(p(n)) \le d \cdot (S^{\mathrm{RAM}}(n))^6 b^{aS^{\mathrm{RAM}}(n)}$.

Now we substitute for $S^{\mathrm{RAM}}(n)$ and obtain $\exists b > 0\, \exists d > 0\, \exists c' > 0\, \forall a > 0 : T^{\mathrm{BSP}}(n) \le db^{6\log_b(c'n)} b^{ac'n} \le db^{2ac'n}$. We immediately get the inequality $\forall a > 0 : T^{\mathrm{BSP}}(n) \le (c^n)^a$ which means that the sequential algorithm for $\mathcal{P}$ is sped up more than polynomially. Hence $\mathrm{BSP}(p, g(p), l(p)) \notin \mathcal{C}_1$.

$\mathrm{BSP}(p, g(p), l(p)) \notin \mathcal{C}_2$:

Consider an arbitrary $\mathrm{BSP}(p, g(p), l(p))$ algorithm for solving $\mathcal{P}$, performing at least one superstep and communicating at least one bit, i.e., $H \ge 1$ and $s \ge 1$. Then $T^{\mathrm{BSP}}(n) \ge m(p(n))$. Assume that the BSP machine is in the second class and thus $T^{\mathrm{BSP}}(n) \le d \cdot (S^{\mathrm{RAM}}(n))^b \le d \cdot (c'n)^b$ for some constants $d > 0$ and $b > 0$. As for the number of processors, according to Lemma 3.18 with $\varepsilon = 1$, $\exists c'' > 0 : p \ge c'' \sqrt{\frac{c^n}{n^b}}$. We substitute the number of processors into the BSP time and obtain $\exists c'' > 0\, \exists c > 0\, \exists b > 0\, \forall k > 0 : T^{\mathrm{BSP}}(n) \ge m(p(n)) \ge \log^k \left(c'' \sqrt{\frac{c^n}{n^b}} \right) = (\log c'' + \frac{n}{2}\log c - \frac{b}{2}\log n)^k \ge \left(\frac{n}{4}\log c\right)^k$. We obtained $\forall k > 0 : T^{\mathrm{BSP}}(n) = \omega(n^k)$ which is a contradiction to the assumption $T^{\mathrm{BSP}}(n) \le d \cdot (c'n)^b = O(n^b)$. Thus $\mathrm{BSP}(p, g(p), l(p)) \notin \mathcal{C}_2$.

□

3.3.4 Relation to Weak Parallel Machines

To study the relation of the BSP model to the class of weak parallel machines, a pipelined version of the BSP computer is needed. We augment the definition of the RAM by input/output devices suitable for pipelined computation. The pipelining extension of the RAM can be transformed to the BSP model in a straightforward way. Since individual instructions of a RAM or a BSP computer can take different numbers of time units and the BSP computes — and thus can read input or write output — only during a part of each superstep, we slightly relax the definition of time and period of a pipelined computation (cf. Definition 2.5).

Definition 3.20 *A computation of a pipelined machine on inputs of length n runs in* time *$T(n)$ and with* period *$P(n)$ iff the N-th output is printed after at most $T(n) + (N-1)P(n)$ time units since the beginning of the computation.*

Definition 3.21 (Pipelined RAM) *A pipelined RAM* (pipe-RAM) *is a RAM equipped with additional two dimensional array I of read-only registers and two dimensional array O of write-only registers. The j-th value of the i-th input word w_i can be read from register $I_{i,j}$ and the j-th value of the i-th output is written to register $O_{i,j}$. Special register io_select is used to choose the particular input or output. The initial value of register io_select is 0 and the only operation permitted for io_select is* `inc` *to increment the value of io_select by one. The increment is performed in unit time regardless the value of io_select. The only way*

to access the input is by the instruction `r:=in(i)` *which copies the content of $I_{io_select,i}$ into read-write RAM register r. Only the number of bits of index i, the index of register r, and the value of $I_{io_select,i}$ contribute to the time cost of the* `r:=in(i)` *instruction. Particularly, the time cost of the instruction does not depend on the value of io_select. Similarly, the output is written by the instruction* `out(i,r)` *which copies the content of RAM register r into output register $O_{io_select,i}$. The cost of the instruction depends again only on the number of bits in index i, the index and the value of r, rather than on the value of io_select. The input is read in order $w_1, w_2, \ldots$ and the output is printed in the same order.*

Definition 3.22 (Pipelined BSP) *A pipelined BSP computer* (pipe-BSP) *has the same input/output register arrays as the pipe-RAM, but only the first $O(n^k)$, where $k > 0$ is a constant, processors may access the I/O registers.*

The following theorem (analogous to Theorem 2.32) and its corollaries show that unrestricted pipe-RAM and pipelined BSP computers are too powerful for being a weak parallel machine.

Theorem 3.23 *Every sequential $T^{\mathrm{RAM}}(n)$-time and $S^{\mathrm{RAM}}(n)$-space bounded RAM algorithm can be simulated by a pipelined RAM in time $T^{\mathrm{RAM}}_{\mathrm{pipe}}(n) = 2^{O(n)}T^{\mathrm{RAM}}(n)$, space $S^{\mathrm{RAM}}_{\mathrm{pipe}}(n) = 2^{O(n)} + O(S^{\mathrm{RAM}}(n))$, and period $P(n) = O(n^2)$.*

Proof. The theorem uses the table lookup technique from the proof of Theorem 2.32. The table of results for all possible inputs of size n can be generated during a precomputation phase and stored using $2^{O(n)}$ registers with addresses of $\log(2^{O(n)}) = O(n)$ bits, yielding the total space occupied by the table to be $2^{O(n)}O(n) = 2^{O(n)}$. Additional space $O(S^{\mathrm{RAM}}(n))$ is needed as the workspace for the precomputation. The total time of the precomputation phase is $2^{O(n)}T^{\mathrm{RAM}}(n)$, i.e., $2^{O(n)}$-times repeated sequential algorithm. One input consisting of $O(n)$ values of up to $O(n)$ bits is transformed to index i by multiplication of all the input values in time $O(n^2)$. The index is used to obtain the i-th element of the table containing the output value in time $O(n)$ because the index has $O(n)$ bits. □

Corollary 3.24 *The pipelined RAM does not belong to the class of weak parallel machines.*

Proof. See the Corollaries 2.33 and 2.34. Consider problem $\mathcal{P}$ with TM space complexity $S(n)$ such that $\forall k > 0 : S(n) = \omega(n^k)$. There is a pipe-RAM algorithm for $\mathcal{P}$ working with period $P(n) = O(n^2)$. Assuming pipe-RAM $\in \mathcal{C}_{\mathrm{weak}}$, we obtain a TM algorithm for $\mathcal{P}$ with space complexity $O(n^k)$ for some constant $k > 0$. This is a contradiction to space complexity lower bound $S(n)$ for problem $\mathcal{P}$. □

Corollary 3.25 *The pipelined BSP is not a member of the class of weak parallel machines.*

Proof. The claim follows immediately from the previous corollary, because a RAM algorithm is essentially equivalent to a BSP algorithm which uses only one processor. □

Similarity of Theorems 2.32 and 3.23 indicate that some restriction of the pipelined BSP model would be necessary in order to become a weak parallel machine. We propose a work-preserving BSP computer. Its main idea resembles that of the strictly pipelined parallel TM (see Definition 2.30): a work-preserving BSP cannot benefit from reusing work done on one instance when computing another instance.

Definition 3.26 (Work-preserving BSP) *A pipelined BSP computer is work-preserving, iff the amount of work necessary to process a sequence of N inputs of size n is at least equal to $NW(n)$, where $W(n)$ is the smallest possible work done by a nonpipelined BSP on a single input of size n.*

Theorem 3.27 *Assume that there exists problem $\mathcal{P}$ such that the fastest RAM algorithm for $\mathcal{P}$ has time complexity $T^{\mathrm{RAM}}(n) \geq c^n$ for some constant $c > 0$ and space complexity $S^{\mathrm{RAM}}(n) \leq c'n$ for a constant $c' > 0$. Let $\forall k > 0 : g(p) > \log^k p$. Then there is no constant $k > 0$ an no work-preserving $\mathrm{BSP}(p, g(p), l(p))$ computer — even if potentially unlimited number of processors is allowed — computing with period $P^{\mathrm{BSP}}(n) = O((S^{\mathrm{RAM}}(n))^k)$.*

Proof. The proof is similar to the part "$\notin \mathcal{C}_2$" of the proof on Theorem 3.19. We take an arbitrary pipelined work-preserving $\mathrm{BSP}(p, g(p), l(p))$ algorithm solving $\mathcal{P}$, which communicates at least a polynomial fraction of bit per period per processor. Such assumption is acceptable, beause to exploit all the available processors, information about the input (at least one bit per instance) must reach them. But the input registers are accessible only by the first $O(n^r)$ processors for some constant $r > 0$. Thus some communication is necessary and $P^{\mathrm{BSP}}(n) \geq g(p(n))/n^r$. There is a sequence of N inputs, each of size n, to be processed. It can be done on the work-preserving RAM in time $T^{\mathrm{RAM}}(n, N) = Nt(n) \geq Nc^n$, where $t(n)$ is the time needed to process just one input. The pipelined computation takes time $T^{\mathrm{BSP}}(n) + (N-1)P^{\mathrm{BSP}}(n)$. Assume that $\mathrm{BSP} \in \mathcal{C}_{\mathrm{weak}}$ and thus $P^{\mathrm{BSP}}(n) \leq d.(S^{\mathrm{RAM}}(n))^b \leq d.(c'n)^b$ for some constants $d > 0$ and $b > 0$. We utilize Lemma 3.18 to obtain a lower bound for the number of processors: if $\varepsilon = 1$, then $\exists c'' > 0 : p \geq c'' \sqrt{\frac{N.c^n}{T^{\mathrm{BSP}}(n)+(N-1).P^{\mathrm{BSP}}(n)}}$. Length N of the input sequence can be arbitrarily large, hence $p \geq \lim_{N\to\infty} p \geq c'' \sqrt{\frac{c^n}{P^{\mathrm{BSP}}(n)}} \geq c'' \sqrt{\frac{c^n}{n^b}}$. After substitution of the number of processors into the period we get $\exists c'' > 0 \, \exists c > 0 \, \exists b > 0 \, \exists r > 0 \, \forall k > 0 : P^{\mathrm{BSP}}(n) \geq g(p(n))/n^r \geq \frac{1}{n^r} \log^k \left(c'' \sqrt{\frac{c^n}{n^b}} \right) = \frac{1}{n^r} \left(\log c'' + \frac{n}{2} \log c - \frac{b}{2} \log n \right)^k \geq \frac{1}{n^r} \left(\frac{n}{4} \log c \right)^k$. We obtained $\forall k > 0 : P^{\mathrm{BSP}}(n) = \omega(n^{k-r})$ which is a contradiction to the assumption $P^{\mathrm{BSP}}(n) = O(n^b)$. Hence $\mathrm{BSP}(p, g(p), l(p)) \notin \mathcal{C}_{\mathrm{weak}}$. □

Corollary 3.28 *If the same assumptions as in Theorem 3.27 are valid, the work-preserving $\mathrm{BSP}(p, g(p), l(p))$ computer is not a member of the class of weak parallel machines.*

Proof. Follows directly from the definition of the class $\mathcal{C}_{\mathrm{weak}}$ and from Theorem 3.27. □

Note that any strictly pipelined PTM (which is a member of $\mathcal{C}_{\mathrm{weak}}$) is a work-preserving machine. It could be therefore assumed that a work-preserving BSP is at least as fast as members of class $\mathcal{C}_{\mathrm{weak}}$. But we proved that although the work preservation is a weaker restriction than strictness of pipelined PTM, the work-preserving BSP is too slow for membership in the class of weak parallel machines. On the other hand, an unrestricted BSP computer is too fast. The conclusion is that the BSP model does not fit to the concept of weak parallelism. In the following chapter, we will define an extended model (called decomposable BSP) and we will prove that it is a member of class $\mathcal{C}_{\mathrm{weak}}$ under certain conditions.

Chapter 4

BSP with Communication Locality: Decomposable BSP

The analysis of the BSP model presented in Chapter 3 shows that — in the sense of computational complexity equivalence up to polynomial factors — BSP computers capture a broad range of models of computational devices. Depending on its parameters p, g, and l, the BSP model scales from the first class (sequential) machines to the highly parallel second class models. It is a well known fact that the machines belonging to the second class are physically infeasible when the laws of nature are taken into account [10]. The two basic constraints are the finite speed of light and the lower bound on the size of an elementary logical gate. The Turing machine is feasible according to these two limits. During time T, any real parallel computer can utilize only up to $O(T^3)$ processors, yielding only a polynomial speedup in comparison with the Turing machine. Hence, no physically feasible model can be significantly[1] asymptotically faster than the first class devices.

Although the exponential parallelism of the second class is practically infeasible, the parallel computers with less efficient parallelism, e.g., parallel Turing machines [78, 79], do obey the above mentioned physical constraints. This fact indicates that the class of weak parallel machines (see Section 2.4) contains some realistic parallel machine models. Yet, regardless of the parameter values, BSP computers do not belong to class $\mathcal{C}_{\text{weak}}$, as was proved in Section 3.3.4. The intuitive reason is the lack of a potential to exploit locality in the BSP model.

In many real parallel computers the cost of communication does not depend only on the size of the message sent between two processors, but also on the topology of the underlying network. If the sender and the destination of the message are in some sense "close to each other", communication between them is much faster than communication between a pair of "distant" processors. Many practical parallel algorithms need communication among some processors very often, while other processors communicate only infrequently. If we exploit such *communication locality* so that intensely communicating processes (logical processors) are placed to "nearby" (physical) processors, then the algorithm may run much faster when compared to some random distribution of processes to processors. There are some models which do reflect the communication locality of a parallel computer. Machines consisting of a number of processors connected by a fixed communication network (mesh, hypercube,...) [36, 63, 81], cellular automata [28], and parallel Turing machines [79] are examples of such models. We can use these models to analyze the impact of different mappings of processes

[1] i.e, more than polynomially

to processors on the performance of an algorithm. The pipelined algorithms from Section 2.4 (working on a parallel Turing machine in such a way that data are exchanged only between neighbour cells) are examples of effective algorithms taking advantage of using locality. One parallel communication step is done in a constant time regardless of the machine's size. Should a piece of data be moved from the first to the last cell of the working tape or vice versa, $O(S(n))$ communication steps are needed. Other example is a d-dimensional mesh network, where the communication cost varies between $O(1)$ for neighbour processors and $O(d\sqrt[d]{p})$ for processors in opposite corners of the mesh.

The BSP model intentionally abstracts from the underlying architecture. Such approach allows developement of efficient portable parallel algorithms. Modeling the properties of a computer by only three parameters p, g, and l greatly simplifies design and analysis of algorithms. When the BSP model is implemented on some parallel machine (typically a distributed memory with message passing or shared memory computer), the underlying machine defines the values of the BSP parameters. The concrete values of parameters are obtained either by a theoretical analysis of the BSP machine mapping to the underlying architecture or by an experiment [16]. A good BSP algorithm should be tunable to a wide range of BSP parameter values and thus usable on any implementation of the BSP model [71].

The main disadvantage — from the point of view of time complexity — of the BSP approach is that no structural information about the computer (i.e, the interconnection network topology) is accessible by the algorithm. Hence all permutations of sets of processes mapped to individual processors are equivalent. If we are interested in an upper bound on the time complexity, we must use the pessimistic values of parameters, which cover the case when majority of communication takes the most expensive paths. For the above example of d-dimensional mesh, this yields to both $g(p)$ and $l(p)$ being $\Omega(d\sqrt[d]{p})$.

We propose an extension of the BSP model in this chapter. The extended model is called the *decomposable BSP*, abbreviated[2] as *dBSP*. An algorithm for dBSP works exactly in the same way as a BSP algorithm except that it can state explicitly that no communication is performed among some processors during a part of computation. Two new instructions `split` and `join` are introduced. After a `split`, processors of a dBSP machine are partitioned (decomposed) into clusters. No communication is allowed between processors from different clusters until the partitioning is cancelled by `join`. Processors within a cluster can communicate freely. The values of parameters g and l depend on the size of a cluster, thus smaller clusters yield faster communication. After the definition of dBSP, some elementary dBSP algorithms are described and their performance is compared to their BSP counterparts. Relations of the dBSP model and machine classes is studied and especially membership in the class of weak parallel machines is proved for a broad range of parameters g and l.

4.1 Definition of a Decomposable BSP Computer

We want to augment the BSP model with some mechanism for exploiting potential communication locality, while retaining the abstract view of the underlying architecture described by only three parameters and the bulk synchronous mode of computation. Any BSP algorithm should also run on the extended (dBSP) model without any modification and have the same time complexity. The additional features should only allow speeding up the computation. The key idea of the desing of efficient dBSP algorithms is to identify those parts of a computation,

[2]not DBSP, as the capital letter "D" usually denotes "deterministic" in the complexity theory

where subsets of processors work independently of other subsets, i.e., no messages are sent from one subset (cluster) to another. BSP parameters g and l increase with increasing number of processors (see Definition 3.2). Intuitively, each cluster can be viewed as an independent BSP machine. Hence, communication and synchronization cost in a cluster does not depend on the total number of processors, but only on the number of members of this cluster. Thus, in order to speed up a computation, we try to minimize size of clusters (and thus maximize their number). On the other hand, a pair of processors which have to communicate must belong into the same cluster. Communication patterns keep changing during a computation, therefore there should be a possibility to dynamically adjust the partitioning.

Definition 4.1 (dBSP) *A* Decomposable Bulk Synchronous Parallel Computer *with p processors and communication parameters g and l — denoted as* dBSP(p, g, l) *— is a* BSP(p, g, l) *computer (see Definitions 3.2, 3.3, and 3.4) with the following differences:*

- ***instruction `split`***

  *During a superstep, a processor can issue instruction **`split(i)`**, where $i \in \{0, \ldots, p-1\}$. If a processor calls **`split`**, then all other processors must call **`split`** exactly once in the same superstep. Beginning from the next superstep, the machine is partitioned into clusters (submachines) $C_1, \ldots, C_{cl}$, where cl is the number of different values of i in instructions **`split(i)`**. Processors which specified the same i in the split instruction belong to the same cluster, while different values of i imply different clusters. Communication is restricted to processors in the same cluster. Sending a message from a processor in one cluster to a processor in another cluster is forbidden. The inactive processors are implicitly assigned to cluster 0, i.e., after a split, only an active processor from cluster 0 may activate a new processor.*

  *Cluster C_i can be further recursively decomposed using instruction **`split(j)`**, which assigns the calling processor to subcluster $C_{i,j}$. Partitioning into subclusters is done independently in each cluster, i.e., partitioning of other clusters C_i', $i \neq i'$, in the same superstep is not required and calling **`split(j)`** (with the same j) by processors in C_i, $C_{i'}$, $i \neq i'$, yields two disjoint subclusters $C_{i,j}$ and $C_{i',j}$.*

- ***instruction `join`***

  *Instruction **`join`** called by a processor cancels the last level of decomposition which involves the calling processor. All the processors in all the sibling — originated from the same **`split`** operation — clusters must call **`join`** exactly once in the same superstep. Only one level of join is allowed in a single superstep. If a higher-level join is required, lower-level joins must be performed first.*

  After a join, the machine can be decomposed again. Assignment of processors to clusters may differ from one partitioning to another.

- ***time complexity***

  *Instruction **`split(i)`** contributes $\log i$ time units and **`join`** contributes one time unit to total work W. The* time complexity *of a dBSP computation is*

$$T^{\text{dBSP}} = \sum_{i=1}^{s} \Big(w_i + h_i g(p_i) + l(p_i) \Big) = W + \sum_{i=1}^{s} \Big(h_i g(p_i) + l(p_i) \Big) \, ,$$

where p_i is the size (number of processors) of the largest non-partitioned cluster existing in superstep s.

In the above definition, `split` and `join` do not induce any communication cost, although in a real implementation, they would require some communication. However, its cost could be included in synchronization cost of supersteps $l(p)$. An example of a dBSP partitioning is shown in Figure 4.1. Instructions `split` and `join` are used for a recursive partitioning, merging of clusters, and repartitioning into a different set of clusters.

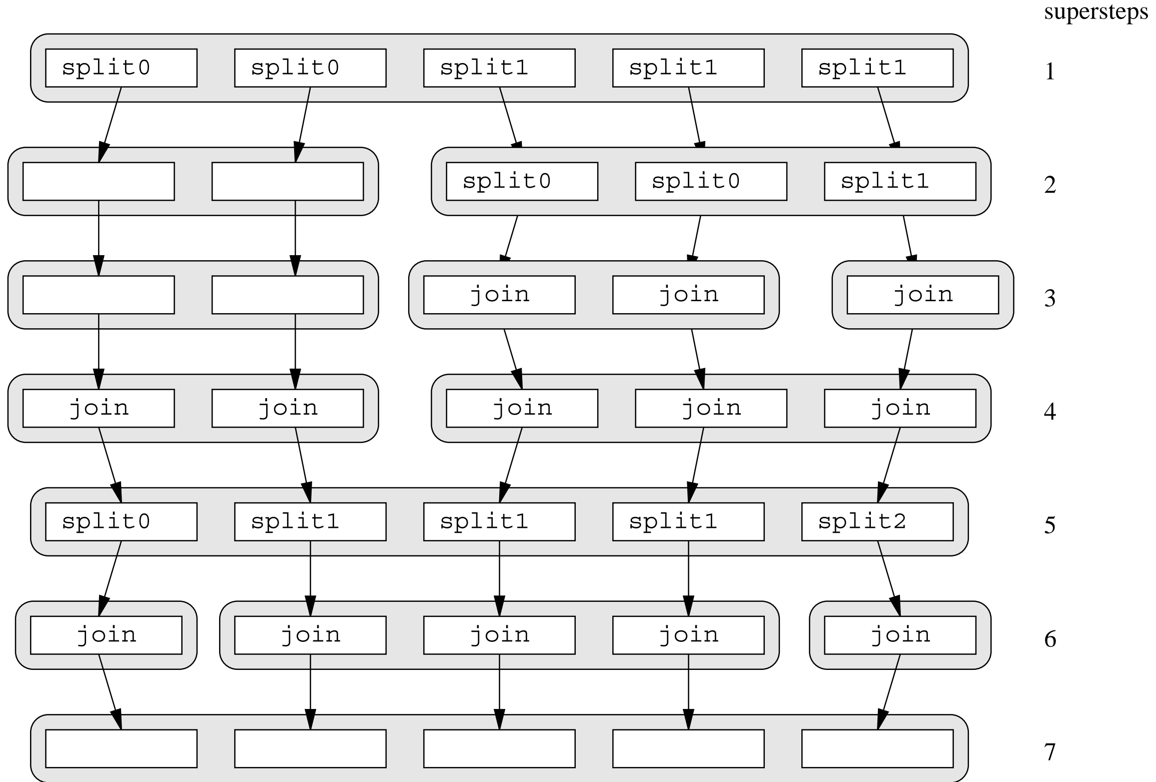

Figure 4.1: Example of dBSP instructions `split(i)` and `join`

4.2 Algorithms for dBSP Computers

In order to show superiority of dBSP computers, BSP algorithms for some basic computational problems, together with their dBSP modifications, are presented in this section. For each algorithm, BSP and dBSP time complexities are compared. We often show only the dBSP algorithms. Their BSP counterparts can be obtained by omitting all `split` and `join` instructions. Most of the dBSP algorithms were obtained from already known BSP algorithms by augmenting them by split and join functionality. The only new algorithms are the two-phase and recursive n-element broadcasting in Section 4.2.2. References to the original BSP algorithms are provided. The uniform cost functions[3] are used throughout this section, since

[3]See the discussion of uniform and logarithmic costs in Section 2.2.

our algorithms do not try to misuse the unit cost of every instruction. General formulae determining the time complexity and the speed-up factor (in comparison with the BSP) are shown as well as results for concrete functions $g(p)$ and $l(p)$. We usually set both functions $g(p)$ and $l(p)$ to be $\Theta(p^a)$ for a constant $0 < a \leq 1$. These values correspond to an implementation of the BSP model by a mesh network of dimension $1/a$. Moreover, $\text{BSP}(p, p^a, p^a)$ belongs to the first machine class (see Theorems 3.13 and 3.14) and we know that any physically feasible computer has to be polynomially time-equivalent to the first class models.

The algorithms are written in a Pascal-like pseudo-code. The same program runs on each processor. The index of a processor is stored in read-only variable *pid*. dBSP communication, synchronization, and partitioning operations are expressed by statements

- **send**(X, i) — send value X to processor i,
- **recv**(X) — place data of a received message into X,
- **barrier** — finish the current superstep,
- **split**(i) — put the calling processor to cluster i,
- **join** — join previously split clusters back into original clusters.

The complexity of an algorithm is usually given up to a multiplicative constant. Quite often, the cost will depend on a tunable parameter k. In such cases we will obtain a complexity estimation in form $T(n, k) = \Theta(f(n, k))$. We are interested in the optimal value of parameter k which minimizes the cost. We look for $k_{\text{opt}}(n)$ such that $T(n, k_{\text{opt}}(n)) = \min_k T(n, k)$. We can often easily find optimal value $k'_{\text{opt}}(n)$ for function $f(n, k)$, e.g., by solving the equation $\frac{\partial f}{\partial k}(n, k'_{\text{opt}}) = 0$. It would be nice if we could obtain value $k_{\text{opt}}(n)$ from $k'_{\text{opt}}(n)$. The following lemma states that this method is correct.

Lemma 4.2 *Assume that* $T(n, k) = \Theta(f(n, k))$. *Let* $f(n, k_{\text{opt}}(n)) = \min_k f(n, k)$ *hold for some function* $k_{\text{opt}}(n)$. *Then* $\min_k T(n, k) = \Theta(T(n, k_{\text{opt}}(n))) = \Theta(f(n, k_{\text{opt}}(n)))$. *The same claim is valid also for* $\max_k f$ *and* $\max_k T$.

Proof. If $T(n, k) = \Theta(f(n, k))$, then there are two positive constants a, b such that $\forall n, k : af(n, k) \leq T(n, k) \leq bf(n, k)$. Necessarily also $af(n, k_{\text{opt}}(n)) \leq T(n, k_{\text{opt}}(n)) \leq bf(n, k_{\text{opt}}(n))$, thus $T(n, k_{\text{opt}}(n)) = \Theta(f(n, k_{\text{opt}}(n)))$.
From $\forall n, k : af(n, k_{\text{opt}}(n)) \leq af(n, k) \leq T(n, k)$ we get $\min_k T(n, k) = \Omega(f(n, k_{\text{opt}}(n)))$.
The inequality $\forall n : \min_k T(n, k) \leq T(n, k_{\text{opt}}(n)) \leq bf(n, k_{\text{opt}}(n))$ yields $\min_k T(n, k) = O(f(n, k_{\text{opt}}(n)))$. The proof for maximum is analogous. □

An alternative way to find $k_{\text{opt}}(n)$ which minimizes $T(n, k)$ works if $f(n, k)$ can be expressed as a sum of a nondecreasing and a nonincreasing function of k plus possibly a part independent on k.

Lemma 4.3 *Let* $T(n, k) = \Theta(f(n, k))$. *Further assume that* $f(n, k) = g(n, k) + h(n, k) + c(n)$, *where* $g(n, k)$ *is a function nondecreasing in* k, $h(n, k)$ *is nonincreasing in* k, *and* $c(n)$ *is independent on* k. *Let* $k_0(n)$ *be such that* $g(n, k_0(n)) = h(n, k_0(n))$. *Then* $\min_k T(n, k) = \Theta(f(n, k_0(n))) = \Theta(T(n, k_0(n)))$.

Proof. The second equality follows directly from the definition of $T(n,k)$. To prove the first equality, we analyze two cases:

$k \le k_0(n) \Rightarrow g(n,k)+c(n) \le g(n,k_0(n))+c(n) = h(n,k_0(n))+c(n) \le h(n,k)+c(n) = \Theta(f(n,k))$, since g is nondecreasing and h is nonincreasing. Hence $T(n,k) = \Theta(h(n,k)+c(n))$. By application of Lemma 4.2 we obtain $\min_k T(n,k) = \Theta(\min_k h(n,k)+c(n)) = \Theta(h(n,k_0(n))+c(n)) = \Theta(f(n,k_0(n)))$.

$k \ge k_0(n)$: Similarly, we have $h(n,k)+c(n) \le h(n,k_0(n))+c(n) = g(n,k_0(n))+c(n) \le g(n,k)+c(n) = \Theta(f(n,k))$ and $\min_k T(n,k) = \Theta(\min_k g(n,k)+c(n)) = \Theta(g(n,k_0(n))+c(n)) = \Theta(f(n,k_0(n)))$.

□

4.2.1 Tree Computations

Parallel algorithms for solving the problems of broadcasting, aggregation, and prefix sums usually have a k-ary tree communication structure [42]. Nodes of the tree correspond to the processors in individual supersteps, edges to the messages sent between processors. Broadcasting is the simplest of all these algorithms.

Algorithm 4.4 *Broadcasting*
Input: value x stored in processor 0
Output: value x stored in every processor

We pick an integer constant $k \ge 2$ and assume that the number of processors is a power of k, i.e., $p = k^n$. Communication within the algorithm is structured according to a k-ary tree.

```
program Broadcast(x,k);
begin
    i := p div k;
    for j := 1 to log_k p do begin
        recv(x); { does nothing if x was not sent to this processor }
        if pid mod(ki) = 0 then
            for m := 1 to k − 1 do
                send(x,pid + mi);
        split(pid div i);
        i := i div 2;
        barrier
    end;
    recv(x); { final receive from the last send }
    barrier
end.
```

The scheme of the algorithm is drawn in Figure 4.2. We analyze the cost for $g(p) = l(p) = p^a$. The algorithm has $\log_k p + 1$ supersteps. The processor with $pid = 0$ sends value x to $k-1$ other processors. During each of subsequent supersteps (except the last one), every processor which obtained x sends it to $k-1$ processors — it is a $(k-1)$-relation. Moreover,

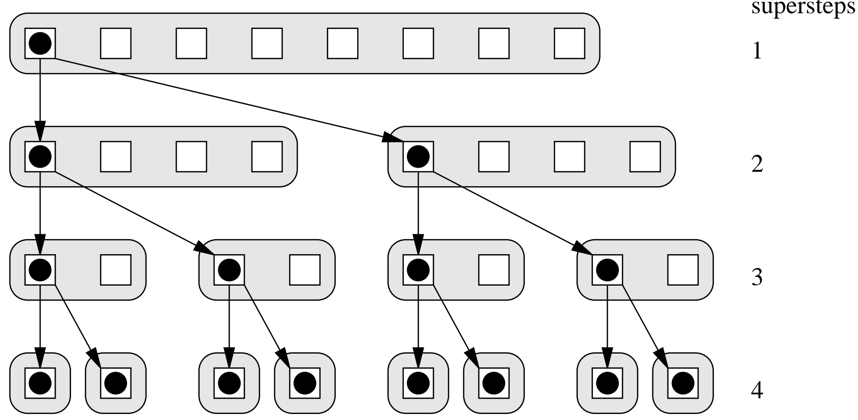

Figure 4.2: dBSP broadcasting algorithm

the dBSP machine splits itself into k submachines of equal size. The BSP and dBSP cost of the algorithm is:

$$T^{\mathrm{BSP}}(p) = \Theta\Big(((k-1)g(p)+l(p))\log_k p + l(p)\Big) = \Theta(kp^a \log_k p) , \quad (4.1)$$

$$T^{\mathrm{dBSP}}(p) = \Theta\left(\sum_{i=1}^{\log_k p} ((k-1)g(k^i)+l(k^i)) + l(1)\right) = \quad (4.2)$$

$$= \Theta\left(\sum_{i=1}^{\log_k p} k^{ai+1}\right) = \Theta(kp^a) .$$

The optimal value of k (minimizing the time cost) is either 2 or 3, hence the dBSP algorithm runs faster by the speedup factor

$$\frac{T^{\mathrm{BSP}}(p)}{T^{\mathrm{dBSP}}(p)} = \Theta\left(\frac{kp^a \log_k p}{kp^a}\right) = \Theta(\log_k p) = \Theta(\log p) . \quad (4.3)$$

Algorithm 4.5 *Aggregation*
Input: value x_i stored in processor p_i, for each $0 \le i \le p-1$
Output: value of $x_0 \circ x_1 \circ \cdots \circ x_{p-1}$, where $\circ$ is an associative and commutative binary operator computable in time $O(1)$, stored in processor 0

The algorithm has a similar tree structure like the broadcasting algorithm, but the values are sent in the opposite direction, from the leaves to the root. The dBSP must first recursively decompose the processors according to the tree structure, then the actual computation is performed and clusters of processors are recursively joined.

```
program Aggregate(x_0,...,x_{p-1});
begin
    { decompose according to a k-ary tree }
    for j := log_k p - 1 downto 1 do begin
        split(pid div k^j);
        barrier
```

```
    end;
    { actual aggregation }
    if pid mod k > 0 then
        send(x, pid − pid mod k);
    join;
    barrier;
    for j := 1 to log_k p do begin
        if pid mod k^j = 0 then begin
            for i := 1 to k − 1 do begin
                recv(y);
                x := x ∘ y
            end;
            if pid mod k^j = 0 ∧ pid mod k^{j+1} > 0 then
                send(x, pid − pid mod k^{j+1})
        end;
        if j < log_k p − 1 then
            join;
        barrier
    end
    { processor 0 holds the result in x }
end.
```

The cost analysis is exactly the same as for broadcasting. Partitioning and then joining increases the execution time by at most constant factor.

Algorithm 4.6 *Prefix sums*
Input: value x_i stored in processor p_i, for each $0 \leq i \leq p-1$
Output: value $x_0 \circ \cdots \circ x_i$ in processor p_i, where $\circ$ is an associative and commutative binary operator computable in time $O(1)$, for each $0 \leq i \leq p-1$

The algorithm is again tree-structured. The actual computation is performed while recursively joining clusters which are created during initial decomposition phase. In each superstep, s maintains the total sum of values x_i seen so far, while only x_i with $i \leq pid$ contribute to x in processor pid.

```
program Prefix(x_0, ... x_{p−1});
    { decompose according to a k-ary tree }
    for j := log_k p − 1 downto 1 do begin
        split(pid div k^j);
        barrier
    end;
    { actual prefix computation }
    s := x;
    for i := 0 to k − 1 do
        if pid mod k ≠ i then
            send([pid, s], pid − pid mod k + i);
    for j := 1 to log_k p do begin
        for i := 0 to k − 2 do begin
            recv([q, y]);
```

```
            s := s ∘ y;
            if q ≤ pid then
                x := x ∘ y;
        end;
        for i := 0 to k − 1 do
            if j < log_k p ∧ pid mod k^(j+1) ≠ pid mod k^j + ik^j then
                send([pid, s], pid − pid mod k^(j+1) + pid mod k^j + ik^j);
        if j < log_k p − 1 then
            join;
        barrier
    end
    { processor i has x_0 ∘ ··· ∘ x_i stored in x and x_0 ∘ ··· ∘ x_(p−1) in s }
end.
```

The cost analysis of broadcasting and aggregation is directly applicable for prefix sums, yielding the same time complexity (up to a constant factor).

4.2.2 Broadcasting and Aggregation of n Elements

The task of the n-element broadcasting algorithm is the following one: there is an array of n values stored in memory of processor 0. The array is to be replicated to local memory of every processor. We could n-times repeat Algorithm 4.4, but a better BSP algorithm exists [42].

Algorithm 4.7 *Broadcast of n values on BSP*
Input: array $X = [x_0, \ldots, x_{n-1}]$ stored in processor 0, such that $n \geq p$
Output: array X stored in every processor

The algorithm runs in three supersteps. In the first superstep, processor 0 splits array X into p chunks of n/p elements and sends each chunk to a different processor. In the second superstep, each processor sends a copy of its chunk to every other processor. Every processor receives all chunks in the third superstep.

```
program BSPnBroadcast(x_0, ..., x_(n−1));
    if pid = 0 then
        for i := 0 to p − 1 do
            send([x_(in/p), ..., x_((i+1)n/p−1)], i);
    barrier;
    recv([y_0, ..., y_(n/p−1)]);
    for i := 0 to p − 1 do
        send([pid, y_0, ..., y_(n/p−1)], i);
    barrier;
    for i := 0 to p − 1 do begin
        recv([q, y_0, ..., y_(n/p−1)]);
        [x_(qn/p), ..., x_((q+1)n/p−1)] := [y_0, ..., y_(n/p−1)]
    end
end.
```

The scheme of the algorithm for broadcasting 3 values to 3 processors is displayed in the upper-left corner of Figure 4.3. The algorithm runs in (obviously optimal) time

$$T^{\mathrm{BSP}}(n, p) = \Theta(ng(p) + l(p)) \; . \tag{4.4}$$

We can obtain an n-element aggregation algorithm by reversing the data flow of Algorithm 4.7. At the beginning, every processor holds an array of n values. The desired output is an array of n elements stored in processor 0 and containing result of aggregation of corresponding input elements from all the processors.

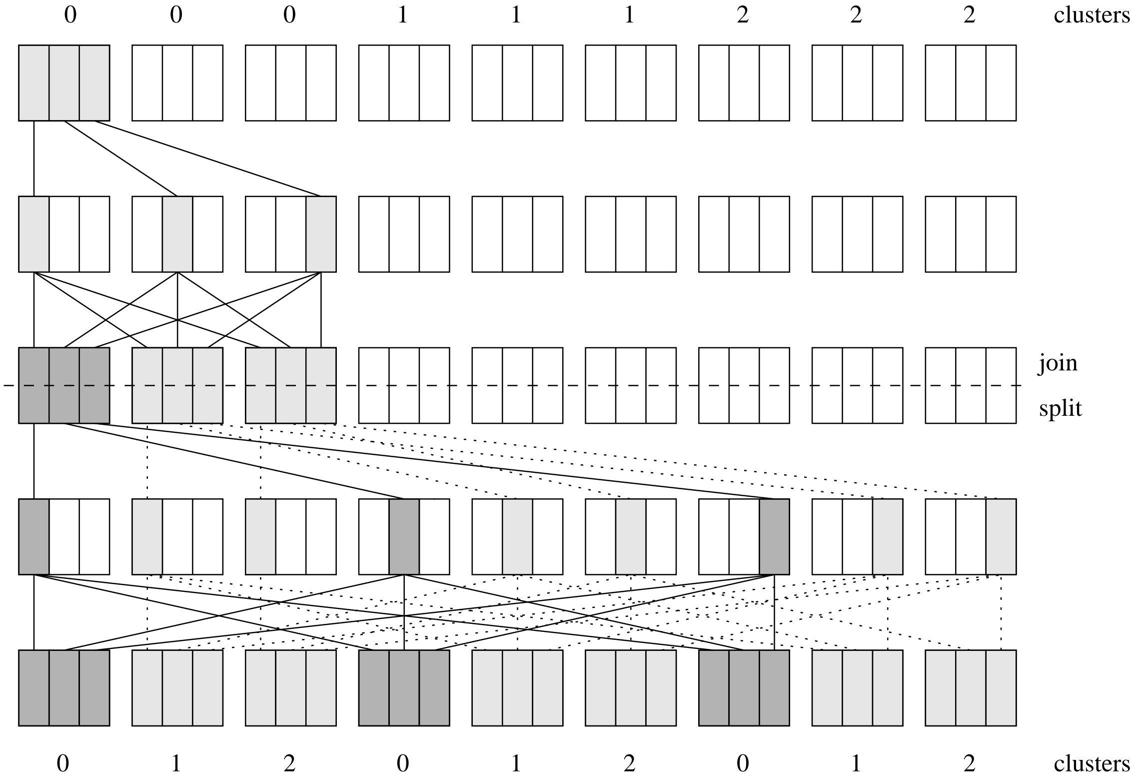

Figure 4.3: Broadcasting of n values on dBSP

Algorithm 4.8 *Aggregation of n values on BSP*
Input: array $X_i = [x_{i,0}, \dots, x_{i,n-1}]$ stored in processor i, where $n \geq p$ and $i \in \{0, \dots, p-1\}$
Output: array $X = X_0 \circ \cdots \circ X_{p-1} = [x_{0,0} \circ \cdots \circ x_{p-1,0}, \dots, x_{0,n-1} \circ \cdots \circ x_{p-1,n-1}]$

In the first superstep, every processor splits its array into p blocks of n/p values and sends the i-th block to the i-th processor. Every processor runs a sequential aggregation algorithm for corresponding values of received array chunks and then sends the resulting block of output array to processor 0. Finally, processor 0 collects the output.

program BSPnAggregate($X_0, \dots, X_{p-1}$);
  for $i := 0$ **to** $p-1$ **do**
    send($[x_{in/p}, \dots, x_{(i+1)n/p-1}, i)$;
  barrier;
  recv($[y_0, \dots, y_{n/p-1}]$);
  for $i := 1$ **to** $p-1$ **do begin**
    recv($[z_0, \dots, z_{n/p-1}]$);
    for $j := 0$ **to** $n/p-1$ **do**
      $y_j := y_j \circ z_j$;

```
    end;
    send([pid, [y_0, ..., y_{n/p-1}]], 0);
    barrier;
    if pid = 0 then
        for i := 0 to p - 1 do begin
            recv([q, [y_0, ..., y_{n/p-1}]]);
            [x_{qn/p}, ..., x_{(q+1)n/p-1}] := [y_0, ..., y_{n/p-1}]
        end
end.
```

The time complexity of the algorithm is

$$T^{\mathrm{BSP}}(n,p) = \Theta(n + ng(p) + l(p)) \tag{4.5}$$

and it is optimal.

Unlike the case of algorithms from Section 4.2.1, we cannot improve the time complexity of Algorithm 4.7 by inserting split and join statements. However, there exists a faster dBSP algorithm for n-element broadcasting. The algorithm runs in two phases, first performing broadcast to $\sqrt{p}$ processors, then to the rest of processors. As Algorithm 4.7 is called in clusters of $\sqrt{p}$ processors, the dBSP algorithm works for $n \geq \sqrt{p}$, instead of $n \geq p$ required by Algorithm 4.7.

Algorithm 4.9 *Broadcast of n values on dBSP*
Input: array $X = [x_0, \ldots, x_{n-1}]$ stored in processor 0, such that $n \geq \sqrt{p}$
Output: array X stored in every processor

The machine is first partitioned into $\sqrt{p}$ clusters of $\sqrt{p}$ processors. Algorithm 4.7 is run in cluster C containing processor 0. Then repartitioning into different $\sqrt{p}$ clusters is performed so that each of the new clusters contains exactly one processor from C. Finally, Algorithm 4.7 is run separately on every cluster.

```
program dBSPnBroadcast(p, x_0, ..., x_{n-1});
    { broadcast to the first √p processors }
    split(pid div √p);
    barrier;
    if pid div √p = 0 then
        BSPnBroadcast(√p, x_0, ..., x_{n-1});
    join;
    barrier;
    { broadcast to remaining processors }
    split(pid mod √p);
    barrier;
    for each cluster in parallel do
        BSPnBroadcast(√p, x_0, ..., x_{n-1});
end.
```

Figure 4.3 shows an example of broadcasting 3 values to 9 processors. Membership of processors in clusters and the moment of repartitioning are marked in the figure. The algorithm's time complexity is given by the cost of repartitioning and of the BSP n-element

broadcasting on a $\sqrt{p}$-processor machine.

$$T^{\mathrm{dBSP}}(n,p) = \Theta\Big(2l(p) + 2T^{\mathrm{BSP}}(n,\sqrt{p})\Big) = \Theta\Big(ng(\sqrt{p}) + l(p)\Big) \ . \tag{4.6}$$

If we assume $l(p) = O(ng(\sqrt{p}))$, i.e., $n = \Omega(l(p)/g(\sqrt{p}))$, and time on BSP according to (4.4), we obtain the speedup

$$\frac{T^{\mathrm{BSP}}(n,p)}{T^{\mathrm{dBSP}}(n,p)} = \Theta\left(\frac{g(p)}{g(\sqrt{p})}\right) \ . \tag{4.7}$$

For example, having $g(p) = l(p) = p^a$, the speedup is

$$\frac{T^{\mathrm{BSP}}(n,p)}{T^{\mathrm{dBSP}}(n,p)} = \Theta(p^{a/2}) \quad \text{if} \quad n = \Omega(p^{a/2}) \ . \tag{4.8}$$

The BSP aggregation (Algorithm 4.8) can be transformed into a dBSP algorithm using the same technique — yielding the same time complexity — as for broadcasting (Algorithms 4.7 and 4.9). Note that if Algorithm 4.9 is run on a BSP computer, its time complexity is $T^{\mathrm{BSP}}(n,p) = \Theta(ng(p) + l(p))$ if $n \geq \sqrt{p}$, while Algorithm 4.7 needs $n \geq p$. The same is true for the dBSP aggregation algorithm run on BSP.

Partitioning into $\sqrt{p}$ clusters of $\sqrt{p}$ processors in Algorithm 4.9 can be done recursively, i.e., instead of calling

BSPnBroadcast($\sqrt{p}, x_0, \ldots, x_n$);

the number of processors is compared to some threshold p_0 and if $p > p_0$ then the dBSP algorithm is called recursively, otherwise the recursion is finished by a call to the BSP algorithm:

```
if p > p0 then
    dBSPnBroadcast(√p, x0, ..., xn)
else
    BSPnBroadcast(√p, x0, ..., xn);
```

The number of processors in clusters gradually decreases by a square root per recursion level: $p, \sqrt{p}, \sqrt{\sqrt{p}} = \sqrt[4]{p}, \sqrt{\sqrt[4]{p}} = \sqrt[8]{p}, \ldots, \sqrt[2^m]{p} = p_0$. There are $\log\log p$ levels of recursion if $p_0 = 2$. It follows from the equation

$$p^{1/q} = p^{1/2^m} = 2 \Leftrightarrow m = \log\log p \Leftrightarrow q = \log p \ . \tag{4.9}$$

We obtain a recurrent formula

$$T^{\mathrm{dBSP}}(n,p) = \begin{cases} 2(l(p) + T^{\mathrm{dBSP}}(n,\sqrt{p})) & \text{if} \quad p > p_0 \ , \\ T^{\mathrm{BSP}}(n,p) & \text{if} \quad p \leq p_0 \ . \end{cases} \tag{4.10}$$

For m levels of recursion, (4.10) yields

$$T^{\mathrm{dBSP}}(n,p) = \sum_{k=1}^{m+1} 2^k l(p^{2/2^k}) + 2^{m+1} ng(p^{1/2^m}) \ . \tag{4.11}$$

This is valid for $n \geq \sqrt[2^m]{p}$, because the BSP algorithm is used in clusters of $\sqrt[2^m]{p}$ processors. Let us assume $g(p) = l(p) = p^a$ for some constant $a > 0$. The equation (4.11) is then transformed to

$$T^{\mathrm{dBSP}}(n,p) = \sum_{k=1}^{m+1} 2^k p^{2a/2^k} + 2^{m+1} n p^{a/2^m} \ . \tag{4.12}$$

We use the inequality for a sum of finite geometric series

$$2p^a = a_1 \leq \sum_{k=1}^{m+1} 2^k p^{2a/2^k} = \sum_{k=1}^{m+1} a_k \leq 2a_1 = 4p^a \ , \tag{4.13}$$

which holds if $\forall k \in \{1,\ldots,m\} : a_{k+1}/a_k \leq 1/2$. This condition determines the permitted interval for values of k and thus the value of m:

$$\begin{aligned} \frac{a_{k+1}}{a_k} = \frac{2^{k+1} p^{2a/2^{k+1}}}{2^k p^{2a/2^k}} &\leq \frac{1}{2} \\ k &\leq \log\log p + \log a - 1 \ . \end{aligned} \tag{4.14}$$

Now we can substitute (4.13) and $m = \log\log p + \log a - 1$ into (4.12) and get

$$T^{\mathrm{dBSP}}(n,p) = \Theta(p^a) + 2^{\log\log p + \log a - 1 + 1} n p^{a/2^{\log\log p + \log a - 1}} = \Theta(p^a) + anp^{2/\log p} \log p \ . \tag{4.15}$$

Finally, (4.9) is used and we obtain the result for $n \geq 2$

$$T^{\mathrm{dBSP}}(n,p) = \Theta(p^a) + 4an\log p = \Theta(p^a + n\log p) \ . \tag{4.16}$$

The speedup in comparison with the BSP algorithm is

$$\frac{T^{\mathrm{BSP}}(n,p)}{T^{\mathrm{dBSP}}(n,p)} = \Theta\left(\frac{p^a}{\log p}\right) \tag{4.17}$$

for $g(p) = l(p) = p^a$, where $a > 0$ is a constant, and $n = \Omega(p^a/\log p)$.

The recursive broadcasting algorithm can be also used on BSP to allow the number of broadcasted values n to be less than p. Formula (4.11) will be slightly changed to

$$T^{\mathrm{BSP}}(n,p) = \sum_{k=1}^{m+1} 2^k l(p) + 2^{m+1} n g(p) = \Theta\Big(2^m (ng(p) + l(p))\Big) \ . \tag{4.18}$$

As $n \geq p^{1/2^m}$ must hold, m is bounded by $m \geq \log\log p - \log\log n$. Note that for n constant (independent on p), we have $T^{\mathrm{BSP}}(n,p) = \Theta((ng(p) + l(p))\log p)$, which is exactly the time complexity of Algorithm 4.4 for broadcasting a single value.

4.2.3 Dense Matrix Multiplication

We present two ways of dBSP parallelization of the standard $\Theta(n^3)$ sequential matrix multiplication algorithm based on the BSP algorithms from [52]. The first (2-dimensional) partitioning method uses a straightforward splitting of the input data, but the second (3-dimensional) algorithm is faster, because it uses $n/p^{2/3}$-relations instead of $n/p^{1/2}$-relations in the 2D algorithm.

Algorithm 4.10 *Matrix multiplication (2D partitioning)*
Input: two $n \times n$ matrices A and B
Output: $n \times n$ matrix $C = AB$

Matrices A and B are both partitioned into $\sqrt{p} \times \sqrt{p}$ equally sized blocks. Hence, the number of processors used during the computation is $p \leq n^2$. Processor $p_{i,j}$ holds blocks $A_{i,j}$, $B_{i,j}$ and computes block $C_{i,j}$. The BSP computation runs in 2 supersteps. First, every processor $p_{i,j}$ sends $A_{i,j}$ to all $p_{i,k}$ and $B_{i,j}$ to $p_{k,j}$, for all $k \in \{0, \ldots, \sqrt{p}-1\}$. Using the received blocks, $p_{i,j}$ computes $C_{i,j} = \sum_k A_{i,k} B_{k,j}$ in the second superstep. The dBSP modification of this algorithm has 5 supersteps. In the first one, the machine is partitioned so that every row of $\sqrt{p}$ processors belongs to a separate cluster. The second superstep includes exchanging of A blocks in rows and the join operation. Then, a similar partitioning into columns and distributing B blocks is performed in the next 2 supersteps. Finally, blocks of C are computed. Figure 4.4 shows the broadcasting of blocks in clusters and the final computational superstep. A partitioning superstep occurs before each broadcast.

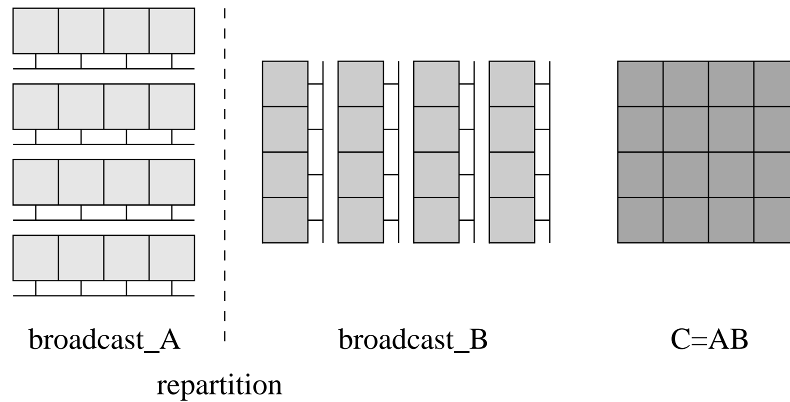

Figure 4.4: Matrix multiplication on dBSP with 2-dimensional partitioning

program MatrixMult2(A, B);
  { $a = A_{i,j}$, $b = B_{i,j}$ stored in processor $i\sqrt{p} + j$ }
  $i := pid$ div $\sqrt{p}$;
  $j := pid$ mod $\sqrt{p}$;
  split(i);
  barrier;
  for $k := 0$ **to** $\sqrt{p} - 1$ **do**
    send$([j, a], i\sqrt{p} + k)$;
  join;
  barrier;
  for $k := 0$ **to** $\sqrt{p} - 1$ **do begin**
    recv$([\hat{\jmath}, \hat{a}])$;
    $\bar{a}_{\hat{\jmath}} := \hat{a}$
  end;
  split(j);
  barrier;

```
    for k := 0 to √p − 1 do
        send([i, b], k√p + j);
    join;
    barrier;
    for k := 0 to √p − 1 do begin
        recv([î, b̂]);
        b̄_î := b̂
    end;
    c := 0;
    for k := 0 to √p − 1 do
        c := c + ā_k b̄_k;
    { c = C_{i,j} is stored in processor i√p + j }
end.
```

Supersteps 1, 2, and 4 of the dBSP algorithm are performed together in the first superstep of the BSP algorithm. The second BSP superstep contains dBSP supersteps 3 and 5. The time complexity is

$$T^{\mathrm{BSP}}(n,p) = \Theta\left(\frac{n^3}{p} + \frac{n^2}{\sqrt{p}} g(p) + l(p)\right) , \quad (4.19)$$

$$T^{\mathrm{dBSP}}(n,p) = \Theta\left(\frac{n^3}{p} + \frac{n^2}{\sqrt{p}} g(\sqrt{p}) + l(p) + l(\sqrt{p})\right) . \quad (4.20)$$

Now let as assume $g(p) = l(p) = p^a$ and $p = n^b$ for some constants $0 < a \leq 1$ and $0 < b \leq 2$. We obtain

$$T^{\mathrm{BSP}}(n) = \Theta\left(n^{3-b} + n^{2+ab-b/2}\right) , \quad (4.21)$$

$$T^{\mathrm{dBSP}}(n) = \Theta\left(n^{3-b} + n^{2+ab/2-b/2}\right) . \quad (4.22)$$

The speedup values according to relation between parameters a and b are summarized in the following table:

	$0 < b \leq 2/(2a+1)$	$2/(2a+1) < b < 2/(a+1)$	$2/(a+1) \leq b \leq 2$
$T^{\mathrm{BSP}}(n)$	$\Theta(n^{3-b})$	$\Theta(n^{2+ab-b/2})$	$\Theta(n^{2+ab-b/2})$
$T^{\mathrm{dBSP}}(n)$	$\Theta(n^{3-b})$	$\Theta(n^{3-b})$	$\Theta(n^{2+ab/2-b/2})$
$T^{\mathrm{BSP}}(n)/T^{\mathrm{dBSP}}(n)$	$\Theta(1)$	$\Theta(n^{ab+b/2-1}) > \omega(1)$	$\Theta(n^{ab/2})$

Algorithm 4.11 *Matrix multiplication (3D partitioning)*
Input: two $n \times n$ matrices A and B
Output: $n \times n$ matrix $C = AB$

The algorithm uses symmetric partitioning of array V of n^3 elementary products $V_{i,k,j} = A_{i,k}B_{k,j}$. Every processor processes a cubic block containing $n/\sqrt[3]{p} \times n/\sqrt[3]{p} \times n/\sqrt[3]{p}$ elements. Thus, the number of exploitable processors is bounded by $p \leq n^3$. Input matrices A, B and

output matrix C can be viewed as projections of V to the coordinate planes (see Figure 4.5). All three matrices are partitioned into square blocks of size $n/\sqrt[3]{p} \times n/\sqrt[3]{p}$. Each block of the matrices A or B is broadcasted to $\sqrt[3]{p}$ processors. Then array V is computed and $C = \sum_k A_{i,k} B_{k,j} = \sum_k V_{i,k,j}$ is obtained by running the aggregation algorithm.

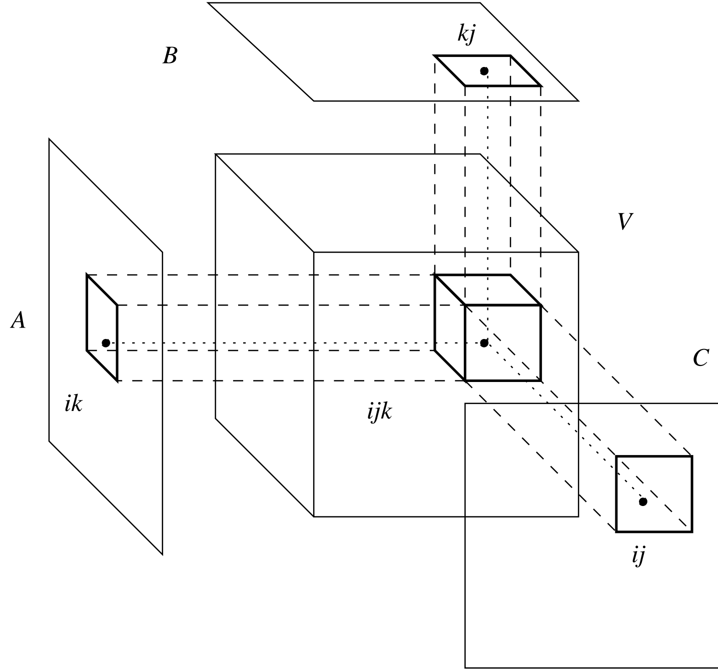

Figure 4.5: Matrix multiplication on dBSP with 3-dimensional partitioning

program MatrixMult3(A, B);
  $q := \sqrt[3]{p}$;
  $b := n/q$;
  $i := pid \bmod q$;
  $k := (pid \operatorname{div} q) \bmod q$;
  $j := pid \operatorname{div} q^2$;
  { A and B stored in processors with $j = 0$ and $i = 0$, respectivelly }
  split$(i + kq)$;
  barrier;
  for each cluster **in parallel do**
    Broadcast$(A_{[ib,\ldots,(i+1)b-1],[kb,\ldots,(k+1)b-1]})$;
  join;
  barrier;
  split$(k + jq)$;
  barrier;
  for each cluster **in parallel do**
    Broadcast$(B_{[kb,\ldots,(k+1)b-1],[jb,\ldots,(j+1)b-1]})$;
  join;
  barrier;

```
        for î := ib to (i+1)b − 1 do
            for ĵ := jb to (j+1)b − 1 do
                for k̂ := kb to (k+1)b − 1 do
                    V_{î,k̂,ĵ} := A_{î,k̂} B_{k̂,ĵ};
        split(i + jq);
        barrier;
        for each cluster in parallel do
            Aggregate(C_{[ib,...,(i+1)b−1],[jb,...,(j+1)b−1]})
end.
```

Calls to "Broadcast" and "Aggregation" in the above program make use of the algorithms from Section 4.2.2. Clearly the total BSP cost of the algorithm (after removing splits and joins) is

$$T^{\mathrm{BSP}}(n,p) = \Theta\left(\frac{n^3}{p} + T^{\mathrm{BSP}}_{\mathrm{bcast}}\left(\frac{n^2}{p^{2/3}}\,,p\right)\right)\,, \tag{4.23}$$

where $T^{\mathrm{BSP}}_{\mathrm{bcast}}$ covers two invocations of broadcasting and one invocation of aggregation algorithm on a p processor BSP machine. There is p instead of $\sqrt[3]{p}$ in $T^{\mathrm{BSP}}_{\mathrm{bcast}}$, because although $p^{2/3}$ copies of the same algorithm are run independently in $p^{2/3}$ clusters, each having only $p^{1/3}$ processors, the communication parameters g and l depend on the total number of processors in the (non-decomposable) BSP machine. Note that in (4.4), p appears only as a parameter of $g(p)$ and $l(p)$. Substitution of (4.4) into the above formula yields

$$T^{\mathrm{BSP}}(n,p) = \Theta\left(\frac{n^3}{p} + \frac{n^2}{p^{2/3}}g(p) + l(p)\right)\,. \tag{4.24}$$

Optimality of this result is proved in [70].

When run on a dBSP computer, the algorithm runs in time

$$T^{\mathrm{dBSP}}(n,p) = \Theta\left(\frac{n^3}{p} + l(p) + T^{\mathrm{dBSP}}_{\mathrm{bcast}}\left(\frac{n^2}{p^{2/3}}, \sqrt[3]{p}\right)\right)\,. \tag{4.25}$$

Any of BSP broadcast (4.4), dBSP one-level broadcast (4.6), or dBSP m-level recursive broadcast (4.11) can be substituted for $T^{\mathrm{dBSP}}_{\mathrm{bcast}}$. In a special case of $g(p) = l(p) = p^a$ and $p = n^b$ for some constants $0 < a \le 1$ and $0 < b \le 3$, using the $\log\log p$-level broadcast (4.16), we obtain

$$T^{\mathrm{BSP}}(n) = \Theta\left(n^{3-b} + n^{2+ab-2/3b}\right)\,, \tag{4.26}$$

$$T^{\mathrm{dBSP}}(n) = \Theta\left(n^{3-b} + n^{ab} + n^{2-2/3b}\log n\right)\,. \tag{4.27}$$

Various settings of a and b yield speedup as shown in the table:

	$0 < b \le \frac{3}{3a+1}$	$\frac{3}{3a+1} < b < \frac{3}{a+1}$	$\frac{3}{a+1} \le b < 3$	$b = 3$
$T^{\mathrm{BSP}}(n)$	$\Theta(n^{3-b})$	$\Theta(n^{2+ab-2/3b})$	$\Theta(n^{2+ab-2/3b})$	n^{3a}
$T^{\mathrm{dBSP}}(n)$	$\Theta(n^{3-b})$	$\Theta(n^{3-b})$	$\Theta(n^{ab})$	n^{3a}
$\frac{T^{\mathrm{BSP}}(n)}{T^{\mathrm{dBSP}}(n)}$	$\Theta(1)$	$\Theta(n^{ab+b/3-1}) > \omega(1)$	$\Theta(n^{2-2/3b})$	$\Theta(1)$

4.2.4 Simulation of Cellular Automata

A 1-dimensional (1D) cellular automaton (CA) consists of an array of n cells, indexed $0, \ldots, n-1$. Each cell is a finite automaton with set Q of states and transition function $\delta : Q \times Q \times Q \to Q$ such that $\forall i \in \{0, \ldots, n-1\} : q_i^{(t+1)} = \delta(q_{i-1}^{(t)}, q_i^{(t)}, q_{i+1}^{(t+1)})$. It means that the state of a cell in time $t+1$ is derived from the state of this cell and its immediate neighbours in time t. In order to define δ for cells q_0 and q_{n-1}, we put $\forall t : q_{-1}^{(t)} = q_n^{(t)} = \hat{q}$, where $\hat{q}$ is the initial state, i.e., in the beginning of a computation, all cells are in state $\hat{q}$. We will restrict ourselves to 1D CA, although cellular automata of any dimension may be defined. A reader interested in the theory of cellular automata may consult [28].

A sequential RAM algorithm and a parallel dBSP algorithm are presented. The parallel algorithm is based on the RAM algorithm and its structure is the same as in the well known finite difference algorithm [27]. Each processor is assigned a subset of cells. Information about states of cells is periodically exchanged among processors.

Algorithm 4.12 *Simulation of CA on RAM*
Input: number of cells n
Output: final state $Q = [q_0, \ldots, q_{n-1}]$ of the CA

The algorithm straightforwardly initializes an array representing the states of individual cells and periodically updates it according to the transition function.

```
program RAM_CA(n);
    for i := −1 to n do
        q_i := q̂;
    while not in terminal state do begin
        for i := 0 to n − 1 do
            q'_i := δ(q_{i−1}, q_i, q_{i+1});
        q := q'
    end
end.
```

Simulation of a single step, i.e., the body of the "while" loop, runs in time

$$T^{\mathrm{RAM}}(n) = \Theta(n) \ . \tag{4.28}$$

Algorithm 4.13 *Simulation of CA on dBSP*
Input: number of cells n
Output: final state $Q = [q_0, \ldots, q_{n-1}]$ of the CA

The automaton is partitioned into blocks of n/p cells, where the number of processors is $p \leq n$. Every processors is responsible for processing of one block. In each simulation cycle, k steps of the CA are simulated. To evaluate $q_i^{(t)}$, one must know $q_{i-1}^{(t-1)}$, $q_i^{(t-1)}$, and $q_{i+1}^{(t-1)}$. To compute these three values, $q_{i-2}^{(t-2)}, \ldots, q_{i+2}^{(t-2)}$ is needed, and so on. If a processor wants to perform k steps, i.e., to find $q_i^{(t+k)}, \ldots, q_{i+n/p-1}^{(t+k)}$, it has to get values $q_{i-k}^{(t)}, \ldots, q_{i+n/p-1+k}^{(t)}$. Consequently, in every simulation cycle, each processor receives $2k$ values from other processors. To make the communication pattern simpler, we require $k \leq n/p$. Then messages are sent only between processors holding neighbouring blocks. communication is performed in two phases. During the first one, the dBSP machine is partitioned into clusters of c processors and data are exchanged among processor belonging to the same cluster. The second phase consists

of repartitioning and performing the rest of communication between processors, which were in different clusters during the first phase. Note that simulation of more than 1 step in one cycle induces some redundant computation, because the values of $q_{in/p-k+1}, \ldots, q_{in/p+k-2}$ are computed twice (by two neighbouring processors) for $i \in \{1, \ldots, p-1\}$. See Figure 4.6 for a scheme of the algorithm. Dashed lines connect cells with their copies needed in neighbouring processors.

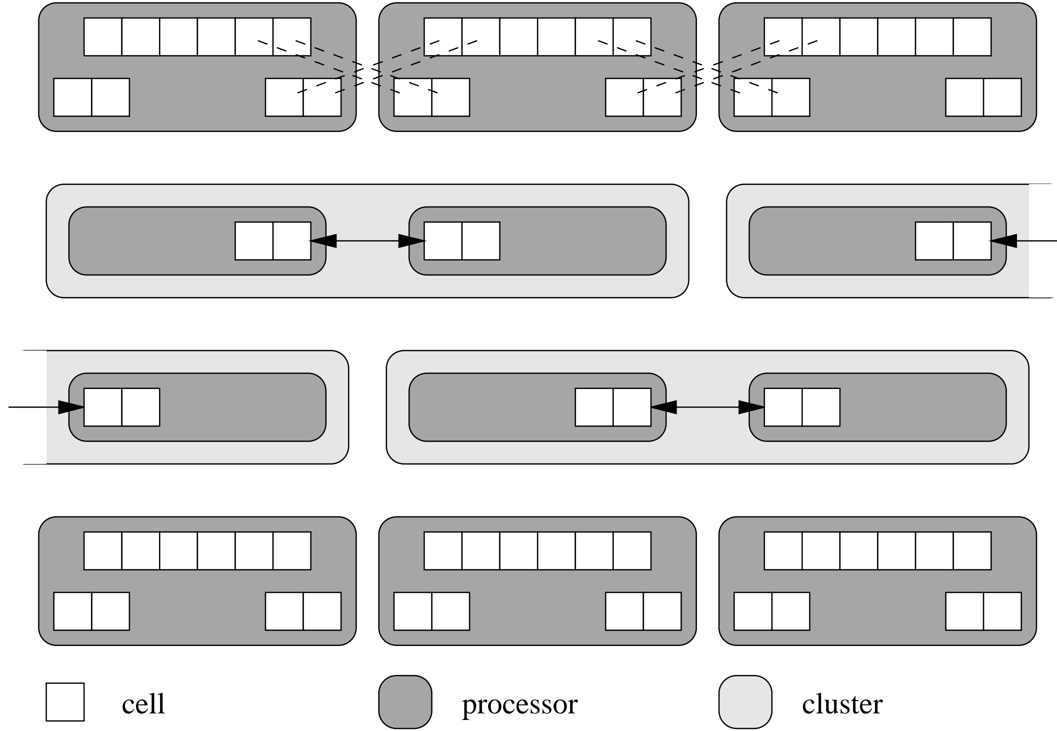

Figure 4.6: Simulation of a cellular automaton on the dBSP model with $n/p = 6$, $k = 2$, and $c = 2$

program dBSP_CA(n, k, c);
    $b := n/p$;
    { cells processed by a processor are stored in array $[q_0, \ldots q_{b-1}]$ }
    for $i := -k$ **to** $b - 1 + k$ **do**
        $q_i := \hat{q}$;
    while not in terminal state **do begin**
        { computation of k steps }
        for $j := k - 1$ **to** 0 **do begin**
            for $i := -j$ **to** $b - 1 + j$ **do**
                $q'_i := \delta(q_{i-1}, q_i, q_{i+1})$;
            $q := q'$;
        end;
        { communicate in clusters of c processors }
        split(pid div c);

```
        barrier;
        if pid mod c ≠ 0 then
            send([q_0, ..., q_{k-1}], pid − 1);
        if pid mod c ≠ c − 1 then
            send([q_{b-k}, ..., q_{b-1}], pid + 1);
        join;
        barrier;
        if pid mod c ≠ c − 1 then
            recv([q_b, ..., q_{b+k-1}]);
        if pid mod c ≠ 0 then
            recv([q_{-k}, ..., q_{-1}]);
        { communicate between processors p_{ci-1} and p_{ci} for 1 ≤ i ≤ pid div c − 1 }
        split((pid + c − 1) div c);
        barrier;
        if pid mod c = 0 ∧ pid > 0 then
            send([q_0, ..., q_{k-1}], pid − 1);
        if pid mod c = c − 1 ∧ pid < p − 1 then
            send([q_{b-k}, ..., q_{b-1}], pid + 1);
        join;
        barrier;
        if pid mod c = c − 1 ∧ pid < p − 1 then
            recv([q_b, ..., q_{b+k-1}])
        if pid mod c = 0 ∧ pid > 0 then
            recv([q_{-k}, ..., q_{-1}]);
    end
end.
```

In the BSP version of the algorithm, the "split" and "join" statements are omitted and simulation of k steps — i.e., the body of the "while" loop — can be done in a single superstep, instead of four. We analyze the average cost of simulating a single step of the cellular automaton. The computational part of every simulation cycle takes time

$$W(n,k) = \Theta\left(\frac{kn}{p} + 2\sum_{i=1}^{k-1} i\right) = \Theta\left(\frac{kn}{p} + k^2\right) . \tag{4.29}$$

Term $\Theta(k^2)$ corresponds to redundant computation. Communication consists of one (BSP) or two (dBSP) $2k$-relations. The whole simulation cycle (simulation of k steps) consists of a constant number of supersteps. In total, we get the average time of a single CA step:

$$\begin{aligned} T^{\mathrm{BSP}}(n,p,k) &= \frac{1}{k}\Theta\Big(W(n,k) + 2kg(p) + l(p)\Big) = \qquad (4.30) \\ &= \Theta\left(\frac{n}{p} + k + g(p) + \frac{l(p)}{k}\right) , \\ T^{\mathrm{dBSP}}(n,p,k,c) &= \frac{1}{k}\Theta\Big(W(n,k) + 4kg(c) + 2l(p) + 2l(c)\Big) = \qquad (4.31) \\ &= \Theta\left(\frac{n}{p} + k + g(c) + \frac{l(p)}{k}\right) . \end{aligned}$$

It is clear that the optimal value of c is $c_{\mathrm{opt}} = \Theta(1)$, because c appears only in nondecreasing term $g(c)$. Hence the dBSP time is

$$T^{\mathrm{dBSP}}(n,p,k) = \Theta\left(\frac{n}{p} + k + \frac{l(p)}{k}\right) . \tag{4.32}$$

Lemma 4.3 can be applied to obtain the optimal values of remaining freely tunable parameters p and k. We only have to obey the restrictions $1 \le p \le n$ and $1 \le k \le n/p$.

$$k = \frac{l(p)}{k} \Rightarrow k_{\mathrm{opt}}(n,p) = \Theta\left(\sqrt{l(p)}\right) . \tag{4.33}$$

Substitution of k_{opt} into (4.30) and (4.32) yields

$$T^{\mathrm{BSP}}(n,p) = \Theta\left(\frac{n}{p} + g(p) + \sqrt{l(p)}\right) , \tag{4.34}$$

$$T^{\mathrm{dBSP}}(n,p) = \Theta\left(\frac{n}{p} + \sqrt{l(p)}\right) . \tag{4.35}$$

The dBSP algorithm is asymptotically faster than the BSP algorithm with the same number of processors iff $g(p) = \omega((n/p) + \sqrt{l(p)})$. If $g(p) = l(p) = p^a$ for a constant $a > 0$, then $k_{\mathrm{opt}}(n,p) = \sqrt{p^a}$ and

$$T^{\mathrm{BSP}}(n,p) = \Theta\left(\frac{n}{p} + p^a\right) , \tag{4.36}$$

$$T^{\mathrm{dBSP}}(n,p) = \Theta\left(\frac{n}{p} + p^{a/2}\right) . \tag{4.37}$$

By utilizing Lemma 4.3 once more the optimal number number of processors is obtained

$$\text{BSP:} \qquad \frac{n}{p} = p^a \Rightarrow p_{\mathrm{opt}}(n) = \sqrt[a+1]{n} , \tag{4.38}$$

$$\text{dBSP:} \qquad \frac{n}{p} = p^{a/2} \Rightarrow p_{\mathrm{opt}}(n) = \sqrt[a+2]{n^2} , \tag{4.39}$$

which results in the optimal BSP and dBSP costs

$$T^{\mathrm{BSP}}(n) = \Theta(n^{a/(a+1)}) , \tag{4.40}$$

$$T^{\mathrm{dBSP}}(n) = \Theta(n^{a/(a+2)}) . \tag{4.41}$$

The only remaining thing is to check that $k_{\mathrm{opt}} \le n/p_{\mathrm{opt}}$. For BSP we have

$$\Theta\left(\sqrt{n^{a/(a+1)}}\right) = \Theta(\sqrt{p^a}) = k_{\mathrm{opt}} \le \frac{n}{p_{\mathrm{opt}}} = \Theta(n^{a/(a+1)}) \tag{4.42}$$

and for dBSP

$$\Theta(n^{a/(a+2)}) = \Theta(\sqrt{p^a}) = k_{\mathrm{opt}} \le \frac{n}{p_{\mathrm{opt}}} = \Theta(n^{a/(a+2)}) . \tag{4.43}$$

The resulting speedup is then

$$\frac{T^{\mathrm{RAM}}(n)}{T^{\mathrm{BSP}}(n)} = \Theta\left(\frac{n}{n^{a/(a+1)}}\right) = \Theta(\sqrt[a+1]{n}) , \tag{4.44}$$

$$\frac{T^{\mathrm{RAM}}(n)}{T^{\mathrm{dBSP}}(n)} = \Theta\left(\frac{n}{n^{a/(a+2)}}\right) = \Theta(\sqrt[a+2]{n^2}) , \tag{4.45}$$

$$\frac{T^{\mathrm{BSP}}(n)}{T^{\mathrm{dBSP}}(n)} = \Theta\left(\frac{n^{a/(a+1)}}{n^{a/(a+2)}}\right) = \Theta\left(n^{\frac{a}{(a+1)(a+2)}}\right) . \tag{4.46}$$

4.3 Simulations of BSP and dBSP Computers

We have already shown examples of algorithms which run asymptotically faster on the dBSP computer than on the BSP. Now we analyze relations between BSP and dBSP models from a general point of view. The basic question about dBSP is how much can be a BSP algorithm sped up when it is ported to the dBSP computer. The starting point is the trivial limit of possible speedup caused by transition from BSP to dBSP.

Theorem 4.14 *Assume that algorithm $\mathcal{A}$ runs in time $T^{\text{BSP}} = W + Hg(p) + sl(p)$ on a* $\text{BSP}(p, g(p), l(p))$ *computer. Then $\mathcal{A}$ will run in time T^{dBSP} on the* $\text{dBSP}(p, g(p), l(p))$ *computer, where*

$$W + Hg(2) + (s-1)l(2) + l(p) \leq T^{\text{dBSP}} \leq T^{\text{BSP}} = W + Hg(p) + sl(p) \ .$$

Proof. The BSP and dBSP versions of algorithm $\mathcal{A}$ differ only in the instructions "split" and "join". As any BSP algorithm is also the dBSP algorithm (with the same time complexity), dBSP is always at least as good as BSP. The only parts of the execution time which can be improved are the factors $g(p)$ and $l(p)$. The largest possible amount of time is saved if the machine is partitioned into clusters of size 2 and all the subsequent computation runs in these clusters. □

An immediate consequence of the theorem is the fact that decomposability could not improve a BSP machine with constant parameters g and l.

Corollary 4.15 *Consider a computer* $\text{BSP}(p, G, L)$*, where G and L are constants. Then for any* $\text{dBSP}(p, g(p), l(p))$ *algorithm $\mathcal{A}$ holds $T_{\mathcal{A}}^{\text{BSP}} = O(T_{\mathcal{A}}^{\text{dBSP}})$.*

An improvement of Theorem 4.14 is possible for BSP algorithms which use only a specific class of h-relations, namely the (h, q)-relations with $q = 1$.

Definition 4.16 *The communication pattern in a particular superstep of a BSP or dBSP computation is an (h, q)-relation, iff each processor sends or receives up to h messages in total to or from at most q processors.*

Now let us consider an algorithm which uses only $(h, 1)$-relations, i.e., each processor communicates with at most one other processor in every superstep. This algorithm can be simulated by a dBSP machine with communication performed only in clusters of two processors, i.e., the dBSP algorithm runs in time $T^{\text{dBSP}} = O(W + Hg(2) + sl(p))$. First we analyze the special case when in a single superstep, a processor can either send or receive data, but not both. Generalization for any $(h, 1)$-relation follows.

Lemma 4.17 *Let the sets of sending and receiving processors be disjoint in an $(h, 1)$-relation. Then this $(h, 1)$-relation can be performed by the* $\text{dBSP}(p, g(p), l(p))$ *computer in time $T^{\text{dBSP}} = \Theta(h + l(p))$.*

Proof. The $(h, 1)$-relation is performed in two superstep. In the first superstep, each processor wanting to send data to other processor q executes instruction `split`(q). Processors which are not senders call `split`(pid). After the barrier synchronization, the machine is decomposed into clusters of 2 processors. The partitioning fulfils the condition that if data are to be sent from processor q_s to processor q_r, then q_s and q_r belong to the same cluster. Thus, the communication can be finished in the second superstep. The total cost of these two supersteps is $T^{\text{dBSP}} = \Theta(l(p) + hg(2) + l(2)) = \Theta(l(p) + h)$, because $g(2)$ and $l(2)$ are constants. □

Theorem 4.18 *Any* $(h, 1)$*-relation can be performed by the* $\mathrm{dBSP}(p, g(p), l(p))$ *computer in time* $T^{\mathrm{dBSP}} = \Theta(h + l(p))$.

Proof. The partitioning scheme from the previous proof cannot be used here because of ambiguity in the processors which are senders and receivers simultaneously. Nevertheless, the communication can be divided into 6 phases. In the first phase, all messages which have a source processor with $pid < p/2$ and destination $pid \geq p/2$ are sent. Likewise, the messages with source $pid \geq p/2$ and destination $pid < p/2$ are sent in the second phase. During the next two phases, remaining messages are exchanged so that each message with sender $pid < p/2$ is transferred to processor $pid + p/2$ and messages from $pid \geq p/2$ go to $pid - p/2$. In the beginning of the fifth phase, all the remaining messages are either stored in processors with $pid < p/2$ and should be moved to a processor having $pid \geq p/2$ or vice versa. Thus the last two phases do the same work as the first two. All 6 phases meet the assumptions of Lemma 4.17. Hence the total time needed by the $(h, 1)$-relation is $T^{\mathrm{dBSP}} = \Theta(6(l(p) + h))$. □

The technique from Theorem 4.18 cannot be used for general (h, q)-relations with $q > 1$. If a processor sends data to more than one of the other processors, then it cannot choose the cluster as the index of the destination processor. Moreover, even for an $(h, 2)$-relation with disjoint sets of sending and receiving processors, the required cluster size could be equal to the total number of processors if the communication should be done in a single superstep. Consider a $(2, 2)$-relation such that every processor with index $0 \leq i < p/2$ sends messages to processors $p/2 + i$ and $p/2 + (i + 1) \bmod (p/2)$. However, every h-relation can be transformed into a sequence of h 1-relations. If an (h, q)-relation is balanced, i.e., h messages are spread across q processors equally, it can be transformed into q (h/q)-relations.

Lemma 4.19 *Any* h*-relation performed in a single superstep can be replaced by* h *supersteps with* 1*-relations.*

Proof. We construct an h-regular bipartite graph G with $2p$ vertices and hp edges. Each processor $i \in \{0, \ldots, p - 1\}$ will be represented by two vertices s_i and r_i in G. If a message should be sent from processor i to processors j then there will be edge (s_i, r_j). Now each vertex has degree at most h. If there is vertex s_i with degree less than h then there is necessarily also vertex r_j with degree less than h and we can add edge (s_i, r_j). Adding an edge can be repeated until all the vertices have degree exactly h. A corollary of the Hall's theorem claims that every h-regular graph contains a 1-factor, see, e.g., [25]. A 1-factor in G corresponds to a 1-relation. After removing the edges of the 1-factor, the remaining graph is $(h - 1)$-regular. Repeating this procedure h-times decomposes the h-relation into a set of h 1-relations. □

Definition 4.20 *A* balanced (h, q)-relation *is such an* (h, q)*-relation in which the number of messages sent from a processor to each destination processor is at most* h/p.

Theorem 4.21 *Any balanced* (h, q)*-relation can be performed by the* $\mathrm{dBSP}(p, g(p), l(p))$ *computer in time* $T^{\mathrm{dBSP}} = O(h + ql(p))$.

Proof. All the messages between every pair of processors are replaced by a single message. The result is a q-relation, further decomposed into q 1-relations according to the previous lemma. Every message represents up to h/q messages. The original (h, q)-relation was transformed into q $(h/q, 1)$-relations. The total time is obtained by q-times applying Theorem 4.18. □

Applicability of Theorem 4.21 is limited because in a general case, partitioning of a bipartite graph into 1-factors must be computed on-line by a distributed algorithm. However, the situation is simplified if the dBSP algorithm is *communication-oblivious*, i.e., the communication and synchronization operations are the same for any input of a given size and can be precomputed off-line. All the algorithms in Section 4.2 are communication-oblivious.

4.4 Variants of dBSP Computers

The original definition of the dBSP model in [7] uses a different semantics of the split operation. After partitioning, each cluster runs independently as a separate dBSP computer until a join is executed. The processors are renumbered so that they obtain new identities $0, \ldots, p_c - 1$ in cluster c containing p_c processors. There is no synchronization among clusters. On the other hand, Definition 4.1 assumes that even after a split, every barrier is a global operation involving all the processors and the processors retain their original pids. We analyze both alternative features separately. For that purpose, we define the renumbering dBSP and the asynchronous dBSP. Then we design mutual simulations of the modified dBSP models, the dBSP from Definition 4.1, and the BSP model. This investigation is motivated by a question of Peter van Emde Boas, who, in a discussion, asked whether the renumbering property of the original dBSP definition could have some side effects by allowing to speed up some algorithms "too much". Algorithm 4.23 gives a positive answer to this question, thus making the respective model less realistic.

4.4.1 Renumbering dBSP

A renumbering dBSP machine changes identifiers of processors (contents of their *pid* registers) after each split or join operation. It reflects better the recursiveness of dBSP than the standard Definition 4.1. After a split, each cluster is again a (renumbering) dBSP computer with processors indexed consecutively beginning from 0.

Definition 4.22 (Renumbering dBSP) *A renumbering dBSP* (rdBSP) *computer is basically the dBSP from Definition 4.1, but after a split operation, all processors get new indices. If cluster c has p_c processors, then its processors will be indexed $0, \ldots, p_c - 1$. The ordering of the new indices is the same as the ordering of the indices before split. After a join operation, the original indices of processors are restored.*

The renumbering dBSP model provides ranking of members of a class of equivalence (i.e., processors in a cluster) for free. This feature can be utilized to obtain some fast algorithms. For example, the OR function can be computed as follows:

Algorithm 4.23 *Fast OR on renumbering dBSP*
Input: value $x_i \in \{false, true\}$ stored in processor p_i for each $0 \le i \le p - 1$
Output: result of $x_0 \vee x_1 \vee \ldots \vee x_{p-1}$ stored in processor 0

The machine is divided into two clusters, one with all $x_i = false$ and the second with all $x_i = true$. If the second cluster is nonempty, the result is $true$, otherwise $false$.

program Renumbering_OR($x_0, \ldots, x_{p-1}$);
  if $x = true$ **then** split(1) **else** split(0);

barrier;
$flag := false$;
if $x = true \wedge pid = 0$ **then** $flag := true$; {**}
join;
barrier;
if $flag$ **then** send$(x, 0)$;
barrier;
if recv(x, q) **then** result is $true$ **else** result is $false$
end.

The renumbering property is used on the line marked with {**} to select a unique processor from all those with $x_i = true$. The algorithm runs in time $T^{\mathrm{dBSP}}(p) = \Theta(g(p) + l(p))$.

Assuming $g(p) = l(p) = O(1)$, the above rdBSP algorithm runs in a constant time. For constant g and l, the standard dBSP and BSP are equally powerful, see Corollary 4.15. If OR was solvable in a constant time on the standard dBSP and hence on the BSP model, then a constant time would suffice also on an EREW PRAM, according to the simulation of the BSP on the EREW PRAM in the proof of Lemma 3.15. But it is proven in [21] that OR requires time at least $\Omega(\log p)$ on the CREW PRAM. Hence, OR can be computed in a constant time on the renumbering dBSP$(p, 1, 1)$, but not on the standard dBSP$(p, 1, 1)$.

Simulations between the standard and the renumbering dBSP models require to maintain the mapping between the global processor indices of the standard dBSP and the local in cluster processor indices of the renumbering dBSP. In many algorithms with a regular communication structure, like all those presented in Section 4.2, this mapping can be easily locally computed in constant time and thus the choice of the standard or the renumbering model has no importance in such case. However, a more complex simulation algorithm is required for a general case, e.g., for the above OR computation.

Theorem 4.24 *A dBSP computer* dBSP$(p, g(p), l(p))$ *can be simulated by a renumbering dBSP computer* rdBSP$(p, g(p), l(p))$ *so that in a situation when the machine is partitioned*[4] *into clusters of maximum size* $p_{\max}$*:*

1. *a split into subclusters of maximum size* p_c *needs time* $O(p_c \log p_c + p_c g(p_c) + l(p_{\max}))$,
2. *a subsequent* h*-relation in a subcluster can be realized in time* $O(h \log p_c + h g(p_c) + l(p_c))$,
3. *other instructions run in the same time on both dBSP and rdBSP.*

Proof.

1. After the split, the simulating renumbering dBSP uses indices of processor which are local to a cluster. On the other hand, the simulated dBSP machine uses global indices of processors. Therefore, the rdBSP machine must maintain the mapping from global to local indices. Array M_c contains pairs of global and local indices $\langle i, i_c \rangle$ for each processor in cluster c. Array M_c is to be stored in each processor belonging to c. Every rdBSP processor remembers its global index i which is the content of its pid register in the beginning of the computation. After the split instruction and the subsequent

[4] If not partitioned (e.g., in the beginning of a computation) then $p_{\max} = p$.

barrier, local index i_c is stored in the processor's *pid* register. All the processors in cluster c send tuple $\langle i, i_c \rangle$ to the processor with local index 0. Processor 0 in every cluster receives p_c tuples $\langle i, i_c \rangle$ and sorts them locally by a sequential sorting algorithm according to global index i. Resulting array M_c is then broadcasted to all processors in the cluster using Algorithm 4.7.

Split and barrier takes time $O(l(p_{\max}))$. Aggregation of the tuples in processor 0 is a p_c relation performed in time $O(p_c g(p_c) + l(p_c))$. Sorting has complexity $O(p_c \log p_c)$. The broadcast is realized in time $O(p_c g(p_c) + l(p_c))$. The total time complexity of the split operation is thus $O(p_c \log p_c + p_c g(p_c) + l(p_{\max}))$ and using Algorithm 4.9 for broadcasting cannot improve it by more than a constant factor.

2. To perform the h-relation, global indices are mapped to local indices by binary searching array M_c. This introduces the term $h \log p_c$.

3. Other instructions are identical on both dBSP and rdBSP.

The same method works also for further recursive partitioning to any level. □

Note that on the dBSP machine, split — the `split` instruction plus the subsequent synchronization — takes $\Theta(l(p_{\max}))$ and h-relation needs time $hg(p_c) + l(p_c)$.

Although Algorithm 4.23 could suggest that renumbering dBSP is strictly more powerful than ordinary dBSP and thus rdBSP could simulate dBSP in the same time, it is not the case. The renumbering machine must maintain the mapping from global to local indices, which slows it down.

Theorem 4.25 *A renumbering dBSP computer* $\mathrm{rdBSP}(p, g(p), l(p))$ *can be simulated by a dBSP computer* $\mathrm{dBSP}(p, g(p), l(p))$ *so that in a situation when the machine is partitioned into clusters of maximum size* $p_{\max}$*:*

1. *a split into subclusters of maximum size* p_c *works in time* $O\Big(p_c \log p_c + p_c g(p_{\max}) + l(p_{\max}) + \sum_{i=1}^{\log_k p_c} ((k-1) p_c g(k^i) + l(k^i))\Big)$ *for some constant* $k > 0$.

2. *a subsequent h-relation in a subcluster can be performed in time* $O(hg(p_c) + l(p_c))$,

3. *other instructions need the same time on the dBSP machine as on the rdBSP.*

Proof.

1. After the split operation, the dBSP computer maintains the mapping from local to global indices. For each cluster c, there is array M_c stored in all the processors of c and containing pair $\langle i_c, i \rangle$ for every processor in cluster c. This is an inversion of the mapping used in Theorem 4.24. After the split, processor i must get new local index i_c. The processor does not know which other processors belong to the same cluster c, therefore indices of all the processors in the cluster cannot be collected in some chosen member processor of the cluster. Instead, indices of processors of cluster c are sent to the processor with index c. This is done before the actual split occurs, because processor c need not be a member of cluster c. Processor c then sorts the indices and assings new local indices $0, \ldots, p_c - 1$. The result is array M_c which is moved to some member of

cluster c. Then the computer is partitioned into clusters and in each cluster c, its M_c is broadcasted to all the processors in c.

Collecting the indices of processors in cluster c by processor c takes time $O(p_c g(p_{\max}) + l(p_{\max}))$. The sequential sorting algorithm costs $O(p_c \log p_c)$. Sending array M_c to a processor from c is performed in time $O(p_c g(p_{\max}) + l(p_{\max}))$. Then $O(l(p_{\max}))$ is spent by partitioning and finally broadcasting according to Algorithm 4.4 runs in time $O(\sum_{i=1}^{\log_k p_c}((k-1)p_c g(k^i) + l(k^i)))$.

2. Given local index i_c, corresponding global index i can be obtained from the i_c-th element of array M_c by a single operation. Hence the h-relation is performed on the dBSP in the same time as on the rdBSP computer.

3. All the other instructions are identical on both rdBSP and dBSP.

The same method can be applied to any further recursive partitioning. □

Here the mapping from local to global indices is implemented by direct addressing into array M_c. Hence the h-relation can be done faster than in Theorem 4.24. Implementing the mapping by simple lookup instead of binary search in the simulation of dBSP on rdBSP would enlarge array M_c from p_c to p elements. Algorithms for efficient n-element broadcasts from Section 4.2.2 cannot be utilized here, because they require that every processor knows all the destinations of the broadcast. The destinations of array M_c are exactly the processors with the global indices contained in M_c. A processor must first obtain whole array M_c to learn of the other processors participating in the broadcast. Thus the less efficient repeated single item broadcasting algorithm is employed.

Algorithms from Theorems 4.24 and 4.25 need additional space of $\Theta(p_c)$ in each processor to store array M_c. Now we show how the renumbering dBSP machine can be simulated by the BSP computer without the need to store the mapping between global and local indices in every processor. Instead, arrays M_c for all clusters are distributed across the processors.

Theorem 4.26 *A renumbering dBSP computer* $\mathrm{rdBSP}(p, g(p), l(p))$ *can be simulated by a* $\mathrm{BSP}(p, g(p), l(p))$ *computer so that:*

1. *a split into subclusters of maximum size* p_c *runs in time* $O((g(p) + l(p)) \log p)$*,*

2. *a subsequent* h*-relation in a subcluster is realized in time* $O(hg(p) + l(p))$*,*

3. *a barrier needs time* $O(l(p))$*,*

4. *all other instructions have the same cost on BSP as on the rdBSP.*

Proof.

1. A global enumeration of all the clusters is maintained during the computation. In a split operation, each processor generates tuple $\langle \bar{c}, c, i \rangle$, where $\bar{c}$ is the global index of the cluster, c is the argument passed to the split instruction, and i is the global index of the processor. The tuples of all the processors are lexicographically sorted, i.e., a processor with greater index will get lexicographically larger tuple. Let the tuple from processor i be held by processor j after sorting. Using the prefix sum algorithm, new global enumeration $\bar{c}'$ of the clusters and ranks r of processors in clusters — the new

local indices — are computed. Resultant tuple $\langle \bar{c}', c, i, r, j \rangle$ is sent to processor i, but is still remembered by processor j. Communication during splitting is shown in Figure 4.7.

Sorting can be done in time $O((1 + g(p) + l(p)) \log p)$ using the BSP sorting algorithm from [65]. Prefix sums are computed in the same time bound by Algorithm 4.6. All other communication (sending of tuples) consists of h-relations with constant h. Thus the total time of a split is $O((g(p) + l(p)) \log p)$.

2. An rdBSP h-relation is performed in 3 supersteps on the BSP. Each of these 3 supersteps consists of a single h-relation. Suppose a message is to be transferred from sender processor s to target processor t. The rdBSP algorithm works with the local indices of the processors. First, the sender processor sends the message to the processor with index j taken from tuple $\langle \bar{c}', c, i, r, j \rangle$ obtained during the simulation of splitting. Tuples belonging to the processors from the same cluster are placed to a continuous range of processors by the sorting. Therefore, the processor holding the tuple corresponding to local index t is $j + (t - s)$. In the second superstep, the message is sent from j to $j + (t - s)$. The tuple stored in processor $j + (t - s)$ determines the global index of receiving processor t which finally gets the message in the third superstep. See Figure 4.8 for an example of communication after a split from Figure 4.7.

3. The simulation machine is BSP, i.e., a barrier needs time $O(l(p))$ instead of $O(l(p_c))$ on the rdBSP computer.

4. All other instructions are the same on both models.

If there are L levels of recursive partitioning, the additional memory needed for storing L tuples is $O(L)$ per processor. □

In the above proof, the tuples needed for communication in one cluster can be located in processors belonging to other clusters. This prevents the possibility of partitioning and the simulating machine is thus BSP, not dBSP.

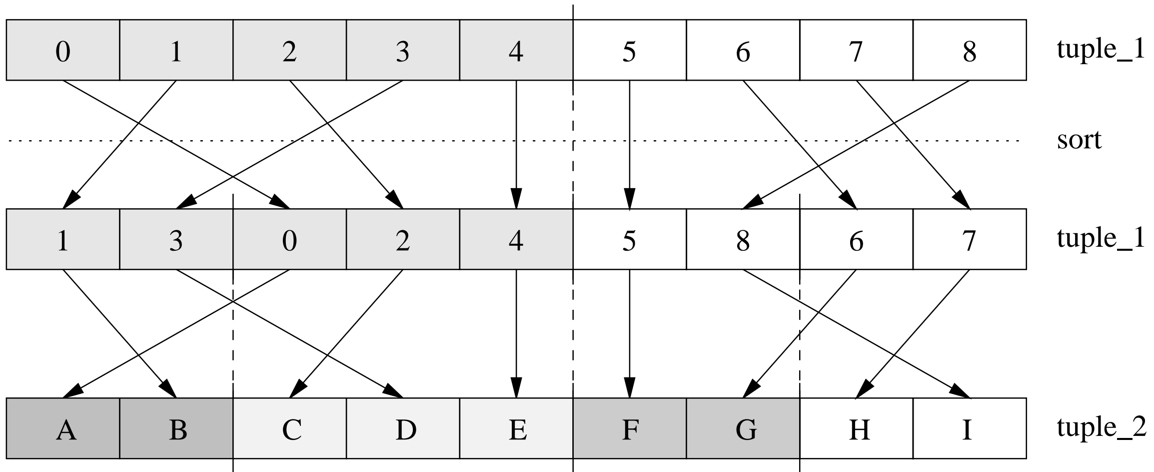

Figure 4.7: Simulation of a rdBSP split on the BSP model

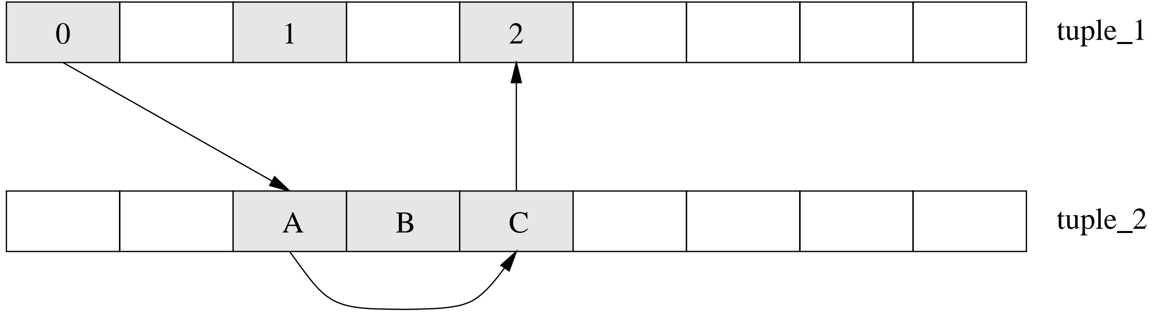

Figure 4.8: Simulation of a rdBSP on the BSP model: sending a message from processor $s = 0$ to processor $t = 2$ in a cluster

4.4.2 Asynchronous dBSP

An asynchronous dBSP computer breaks the synchronization among clusters assumed by Definition 4.1. Such synchronization is not necessary, it does not change result of a computation, because the clusters work independently and they are always (even in asynchronous dBSP) synchronized at the moment of join.

Definition 4.27 (Asynchronous dBSP) *An asynchronous dBSP computer* (adBSP) *is based on the dBSP from Definition 4.1, but after partitioning, clusters work independently of each other. A barrier synchronization is performed separately in each cluster. If a cluster of* p *processors*[5] *is split into* c *subclusters of* $p_0, \ldots, p_{c-1}$ *processors, and subcluster* i *performs* s_i *supersteps with total work* W_i *and communication volume* H_i*, then the execution from the split to the corresponding join runs in time*

$$T^{\text{adBSP}} = \max_{0 \le i < c} T_i^{\text{adBSP}} = \max_{0 \le i < c} \{W_i + H_i g(p_i) + s_i l(p_i)\} \ .$$

The asynchronous dBSP machine can execute many short supersteps in one cluster while a single long superstep is running in some other cluster. On the standard dBSP computer, a long superstep in one cluster delays computation in all other clusters. Recall the dBSP time cost of s supersteps: $T^{\text{dBSP}} = \sum_{j=1}^{s} \max_{0 \le i < c} T_{i,j}^{\text{dBSP}}$. Any standard dBSP algorithm can be executed by an asynchronous dBSP computer in the same or shorter time. The reverse simulation requires ability to partition long supersteps into shorter ones. We exploit the idea used in [9] for simulation of the LogP model on the BSP. The asynchronous dBSP computation is divided into supersteps of the same length. A suitable constant t is chosen. When the length of the current superstep achieves t time units, the superstep is terminated. The remaining work is left to the subsequent supersteps.

There is a technical difficulty in estimation of time spent by a superstep. Consider, for example, the case when $hg(p) = t/2 < phg(p)$ and $l(p) = t/2$. If the superstep is terminated after each processor sends h messages, then we may expect that duration of the superstep will be $hg(p) + l(p) = t$. But all messages sent by all processors could be destined for a single processor. Then a ph-relation is to be realized instead of a h-relation and the superstep will take time $phg(p) + l(p) > t$. In general, if a processor sends a message, it cannot know

[5] or the whole machine in the first level of partitioning

how many messages were sent to the same destination. Hence, the simulation can be slowed down by factor p. To overcome this problem, we restrict our simulation to algorithms with a priori limited communication patterns. We define the *well-behaved* asynchronous dBSP computation, which resembles the notion of well-behaved LogP programs used in [9]. The definition ensures that if a well-behaved superstep is terminated after at most w computational steps and h messages sent per processor then its time complexity is $O(w + hg(p) + l(p))$.

Definition 4.28 *A computation is* well-behaved *iff at any time, when each processor has sent at most h bits in messages since the beginning of the current superstep, no more than $O(h)$ bits share the same destination processor.*

Theorem 4.29 *Let a well-behaved asynchronous dBSP computer be divided into c clusters cointaining from $p_{\min}$ to $p_{\max}$ processors. Further assume that $g(p_{\max}) \leq l(p_{\min})$. Then its computation can be simulated by a standard (synchronous) dBSP machine with slowdown $O(l(p_{\max})/l(p_{\min}))$.*

Proof. The simulating dBSP computer will be synchronized after performing every $l(p_{\min})$ steps. The computational steps are counted and $g(p_{\max})$ time units are charged per message locally in each processor. When value $t_i = w_i + hg(p_{\max})$ in processor i reaches $l(p_{\min})$, the barrier instruction is executed by the processor. The communication has form of $O(h)$-relation, because the algorithm is well-behaved. The assumption $g(p_{\max}) \leq l(p_{\min})$ ensures that at least one message can be sent in time $l(p_{\min})$. On the asynchronous dBSP machine, computation and sending of messages are parts of a superstep. Their cost is at least t_i plus a respective fraction of synchronization time and possibly waiting time for remaining processors in the same cluster to perform the barrier. Hence, we have $T^{\text{adBSP}} \geq t_i = w_i + hg(p_{\max}) \geq l(p_{\min})$. If an adBSP processor performs the synchronization operation before it completes t_i steps, the time cost is still $T^{\text{adBSP}} \geq l(p_{\min})$ due to the duration of the barrier instruction. The dBSP computer simulates the same part of execution in a single superstep, i.e., in time $T^{\text{dBSP}} = O(w_i + hg(p_{\max}) + l(p_{\max})) = O(l(p_{\min}) + l(p_{\max}))$. According to Definition 4.1, the values of g and l depend on the size of the largest cluster. Now we can immediately obtain desired slowdown $T^{\text{dBSP}}/T^{\text{adBSP}} = O(l(p_{\max})/l(p_{\min}))$. □

We have shown that — in some cases — the renumbering and asynchronous dBSP computers provide superior performance when compared to the standard dBSP model. Additional speedup is caused by:

- too powerful renumbering property of rdBSP which can be exploited to obtain some unrealistically fast algorithms, e.g., a constant time OR computation (Algorithm 4.23);
- removing unnecessary synchronization of processors belonging to different clusters in case of adBSP. This is a realistic improvement, because it only makes computation in each cluster independent on the other clusters.

On the other hand, all three models (dBSP, rdBSP, and adBSP) are equivalent for many practical algorithms. Thus it may be viable to use the renumbering asynchronous dBSP model, which captures the recursive nature of decomposable BSP better than the standard dBSP, because each cluster is again an independent radBSP machine with processors idexed consecutively from 0 upwards.

4.5 Membership of dBSP in Machine Classes

Results presented in this section are analougous to those proved for the BSP model in Section 3.3. An important difference is the fact that a properly restricted dBSP machine belongs to the class of weak parallel machines, see Section 4.5.4.

4.5.1 Membership of dBSP in $\mathcal{C}_1$

Theorems 3.10 and 3.13 can be reformulated for the dBSP model. Thus, we prove that if the number of processors p is constant, or communication parameter $l(p)$ is at least $\Omega(p^a)$ for some constant $a > 0$ then the dBSP computer belongs to the first class. Due to reasons explained in Section 2.2 the logarithmic time and space costs are used throughout this section.

Theorem 4.30 *Let the number of processors p be a constant (independent on the input size). Then a* $\text{dBSP}(p, g, l)$ *computer with arbitrary values of parameters g and l is a member of the first machine class.*

Proof. Proof of Theorem 3.10 is directly applicable here. Lemma 3.7 is obviously valid for dBSP. Lemma 3.8 holds for a BSP machine with any parameters, therefore its validity also for the dBSP model follows from Corollary 4.15. □

Theorem 4.31 *Let* $l(p) = \Omega(p^a)$ *for some constant* $a > 0$*. Then for any p and* $g(p)$*, a machine* $\text{dBSP}(p, g(p), l(p))$ *is a member of the first machine class.*

Proof. In the proof of Theorem 3.13, we stated that $T^{\text{BSP}} \geq W + skp^a$. But this fact was only used to obtain the estimation $kp^a \leq T^{\text{BSP}}$. According to Theorem 4.14, we have $T^{\text{dBSP}} \geq l(p) = \Omega(p^a) \geq kp^a$ for some constants $a > 0$, $k > 0$. Thus this proof is identical to that of Theorem 3.13. □

Unlike the previous two theorems, the proof of Theorem 3.14 cannot be exploited to prove its dBSP analog. If $l(p)$ is arbitrary, we may choose $l(p) = L$, i.e., a constant. Theorem 4.18 is applicable to the simulation of the EREW PRAM on the BSP in proof of Lemma 3.15 because it makes use of only 1-relations with disjoint sets of sending and receiving processors. Thus, the simulation fulfils $T^{\text{dBSP}(p,g(p),L)} = O(T^{\text{BSP}(p,1,L)}) = O((T^{\text{PRAM}})^2)$ and the $\text{dBSP}(p, g(p), L)$ is only polynomially slower than the $\mathcal{C}_2$ machine EREW PRAM.

4.5.2 Membership of dBSP in $\mathcal{C}_2$

Theorem 4.32 *Let* $g(p)$ *be arbitrary and* $l(p) = O(\log^b p)$ *for some constant* $b > 0$*. Then a* $\text{dBSP}(p, g(p), l(p))$ *computer with potentially unlimited number of processors is a member of the second machine class.*

Proof. See also the proof of Theorem 3.16. PRAM and dBSP are mutually simulated using Lemma 3.15. The simulation of the EREW PRAM on the dBSP uses the communication scheme from Theorem 4.18 to achieve time of 1-relation $O(g(2) + l(p)) = O(\log^b p)$. The complete simulation is slower by factor $O(l(p)) = O(\log^b p) = O((T^{\text{PRAM}})^b)$. Hence $T^{\text{dBSP}} = O((T^{\text{PRAM}})^{2+b})$. □

4.5.3 Outside $\mathcal{C}_1$ and $\mathcal{C}_2$

The dBSP computers with $l(p)$ between $\omega(\log^k p)$ and $o(p^a)$ remain to be examined. We prove that such machines are neither in $\mathcal{C}_1$ nor in $\mathcal{C}_2$, like their BSP counterparts (see Theorem 3.19). Note that for dBSP, Lemma 3.18 holds unchanged but a stronger version of Lemma 3.17 can be proven:

Lemma 4.33 *Let problem $\mathcal{P}$ have a RAM algorithm with space complexity $S^{\text{RAM}}(n)$. Then there is a* dBSP$(p, g(p), 1)$ *algorithm running in time $T^{\text{dBSP}}(n) \le O((S^{\text{RAM}}(n))^6)$ on $p \le 2^{O(S^{\text{RAM}}(n))}$ processors, for any $g(p)$.*

Proof. The transitive closure algorithm from the proof of Theorem 2.17 (the second part, simulation of the Turing machine on the PRAM) can be run on an EREW PRAM in the same time and with the same number of processors, i.e., $T^{\text{PRAM}}(n) = O((S^{\text{RAM}}(n))^3)$ and $p = 2^{O(S^{\text{RAM}}(n))}$. The algorithm uses $r = 2^{O(S^{\text{RAM}}(n))}$ global registers. The PRAM is simulated on the dBSP by the algorithm from the proof of Lemma 3.15 in time $T^{\text{dBSP}}(n) = O((T^{\text{PRAM}}(n))^2) = O((S^{\text{RAM}}(n))^6)$. Using the communication scheme from Theorem 4.18 together with the assumption $l(p) = 1$ yields a constant communication slowdown factor, regardless of $g(p)$. The number of necessary dBSP processors is $p + r = 2^{O(S^{\text{RAM}}(n))}$. □

Theorem 4.34 *Let us assume that there exists problem $\mathcal{P}$ such that the fastest RAM algorithm for $\mathcal{P}$ has time complexity $T^{\text{RAM}}(n) \ge c^n$ for some constant $c > 0$ and space complexity $S^{\text{RAM}}(n) \le c'n$ for a constant $c' > 0$. Let $g(p)$ be arbitrary and $\forall a, k > 0 : \log^k p < l(p) < p^a$. Then a* dBSP$(p, g(p), l(p))$ *computer with potentially unlimited number of processors is neither a member of the first machine class nor the second class.*

Proof.

dBSP$(p, g(p), l(p)) \notin \mathcal{C}_1$:

¿From Lemma 4.33, we obtain an $\mathcal{M}$ = dBSP$(p, g(p), 1)$ algorithm for $\mathcal{P}$ such that $\exists b > 0 : p = b^{S^{\text{RAM}}(n)}$ and $\exists d > 0 : T^{\mathcal{M}} \le d \cdot (S^{\text{RAM}}(n))^6$. When executed on a dBSP$(p, g(p), l(p))$ computer, the algorithm is slowed down by factor $l(p)$, i.e., $\exists b > 0\ \exists d > 0\ \forall a > 0 : T^{\text{dBSP}}(n) \le d \cdot (S^{\text{RAM}}(n))^6 l(p(n)) \le d \cdot (S^{\text{RAM}}(n))^6 b^{aS^{\text{RAM}}(n)}$.

By substitution of $c'n$ for $S^{\text{RAM}}(n)$ we get $\exists b > 0\ \exists d > 0\ \exists c' > 0\ \forall a > 0 : T^{\text{dBSP}}(n) \le db^{6 \log_b(c'n)} b^{ac'n} \le db^{2ac'n}$. From this the inequality $\forall a > 0 : T^{\text{dBSP}}(n) \le (c^n)^a$ follows directly. Hence, the sequential algorithm for $\mathcal{P}$ is speeded up more than polynomially and dBSP$(p, g(p), l(p)) \notin \mathcal{C}_1$.

dBSP$(p, g(p), l(p)) \notin \mathcal{C}_2$:

Any dBSP$(p, g(p), l(p))$ algorithm for $\mathcal{P}$ must have at least one superstep. Therefore $T^{\text{dBSP}}(n) \ge l(p(n))$. Assume that the dBSP computer belongs to the second class and thus $T^{\text{dBSP}}(n) \le d \cdot (S^{\text{RAM}}(n))^b \le d \cdot (c'n)^b$ for some constants $d > 0$ and $b > 0$. According to Lemma 3.18, if $\epsilon = 1$ then $\exists c'' > 0 : p \ge c'' \sqrt{\frac{c^n}{n^b}}$ holds for the number of processors. Substitution of the number of processors into the dBSP time gives $\exists c'' > 0\ \exists c > 0\ \exists b > 0\ \forall k > 0 : T^{\text{dBSP}}(n) \ge \log^k \left(c'' \sqrt{\frac{c^n}{n^b}}\right) = (\log c'' + \frac{n}{2} \log c - \frac{b}{2} \log n)^k \ge (\frac{n}{4} \log c)^k$. But now we have $\forall k > 0 : T^{\text{dBSP}}(n) = \omega(n^k)$ which is a contradiction to the assumption $T^{\text{dBSP}}(n) \le d \cdot (c'n)^b$. We proved dBSP$(p, g(p), l(p)) \notin \mathcal{C}_2$.

□

4.5.4 Weak Parallel Machines and dBSP

Augmenting the dBSP model with arrays of input and output registers in the same way as we defined pipe-RAM (Definition 3.21) and pipe-BSP (Definition 3.22) yields the *pipelined dBSP computer (pipe-dBSP).* Recall also the modified definition of period (Definition 3.20). It would be nice if pipe-dBSP belonged to the class of weak parallel machines, because it would be an indication that more than a polynomial speedup is possible for some computations in comparison with sequential machines. Unfortunately, this is not the case.

Corollary 4.35 *The pipelined dBSP is not a member of the class of weak parallel machines.*

Proof. Follows directly from Theorem 3.23 and Corollary 3.24, because a RAM algorithm can be run on a dBSP computer using just one processor. □

The work-preserving BSP was introduced in Definition 3.26 as a possible limitation of pipe-BSP computational power. The work-preserving dBSP can be defined similarly. Theorem 3.27 claims that the work-preserving BSP is too slow for being a weak parallel machine if $g(p) > \log^k p$. The proof is based on the natural assumption that at least one bit per period (or per instance) is communicated. Then each period takes at least $g(p(n))$ time units where $p(n)$ is exponential given a linear sequential space complexity $S^{\mathrm{RAM}}(n)$. The period is thus larger than any polynomial $(S^{\mathrm{RAM}}(n))^k$. Such a reasoning does not apply to the dBSP model. We have seen earlier that the factor $g(p)$ can be removed by a suitable partitioning, see, e.g., Theorem 4.18.

We will design an algorithm for simulating a sequential TM computation with space complexity $S(n)$ on a dBSP with period $P(n) = O(S^k(n))$. The algorithm resembles the corresponding pipelined PTM algorithm from Lemma 2.25. We know from Corollary 4.35 that the unlimited dBSP is too powerful for belonging to class $\mathcal{C}_{\mathrm{weak}}$. We reuse the ideas from Section 2.4 and define two restrictions of the dBSP model, namely the limited pipe-dBSP and the strictly pipelined dBSP. In the case of parallel Turing machines, there is a notion of a uniform computation, i.e., cycling on a sequence of identical inputs. The same configuration repeats after each $P(n)$ steps. In the dBSP case, the shortest possible cycle length is a single superstep. But we will see that an efficient computation requires several periods to be packed into one superstep. We overcome this technical complication by definining uniform, limited, and strictly pipelined PRAMs first and then transforming these definitions into the dBSP framework. We prove that any limited or strictly pipelined dBSP computer with period $P(n)$ can be simulated by a Turing machine in space $S(n) = O(P^k(n))$ and consequently we conclude that they are members of the class of weak parallel machines.

Lemma 4.36 *Let $\mathcal{P}$ be a problem for which there is a sequential RAM algorithm running in time $T^{\mathrm{RAM}}(n)$ and space $S^{\mathrm{RAM}}(n)$. Let us have a pipelined $dBSP(p, g(p), l(p))$ computer with arbitrary parameter $g(p)$ and with $l(p) = O(p^k)$ for some constant $k > 0$. Then the pipelined version of $\mathcal{P}$ can be solved by the pipe-dBSP machine in time $T^{\mathrm{dBSP}}(n) = O((T^{\mathrm{RAM}}(n))^{k+1} S^{\mathrm{RAM}}(n))$, space $S^{\mathrm{dBSP}}(n) = O((T^{\mathrm{RAM}}(n))^{k+2})$, with period $P^{\mathrm{dBSP}}(n) = O((S^{\mathrm{RAM}}(n))^2)$, and $p = T^{\mathrm{RAM}}(n)/S^{\mathrm{RAM}}(n)$ processors.*

Proof. Each processor stores q instances of $\mathcal{P}$ in its local memory. A superstep comprises computing the next $T^{\mathrm{RAM}}(n)/p$ steps of the sequential algorithm for all qp unfinished

instances. Then every processor sends its instances to the next processor. The oldest q instances are finished and their output written to the output registers. Simultaneously, q new inputs are read from the input registers.

Every processor uses $O(qS^{\mathrm{RAM}}(n))$ registers. The maximum number of bits in an address is therefore $\log(qS^{\mathrm{RAM}}(n)) = \log q + \log S^{\mathrm{RAM}}(n)$ instead of $\log S^{\mathrm{RAM}}(n)$ in the RAM algorithm. Hence an instruction can be slowed down relatively to its RAM counterpart by factor $\log q$. Communication is done in two phases. The computer is twice repartitioned into pairs of processors. In the first phase, each processor with even *pid* sends its data to the successive odd processor. Data from odd to even processors are sent in the second phase.

¿From time needed by a single computational superstep and its related communication supersteps, i.e., q periods, we obtain the length of the period.

$$
\begin{aligned}
qP^{\mathrm{dBSP}}(n) &= \frac{T^{\mathrm{RAM}}(n)}{p} q\log q + 2l(p) + S^{\mathrm{RAM}}(n) q\log q + \log p + qS^{\mathrm{RAM}}(n)g(2) + 2l(2)\ , \\
P^{\mathrm{dBSP}}(n) &= \frac{T^{\mathrm{RAM}}(n)}{p}\log q + 2\frac{l(p)}{q} + S^{\mathrm{RAM}}(n)\log q + \frac{\log p}{q} + S^{\mathrm{RAM}}(n)g(2) + 2\frac{l(2)}{q}\ .
\end{aligned}
$$

The first term corresponds to computation, the next one to partitioning, then two terms follow due to the cost of send/recv instructions. Communication and synchronization in clusters is covered by the last two terms. Now we use the assumption $l(p) = O(p^k)$. We also put $p = T^{\mathrm{RAM}}(n)/S^{\mathrm{RAM}}(n)$ and $q = (T^{\mathrm{RAM}}(n))^k$. This yields $P^{\mathrm{dBSP}}(n) = O(kS^{\mathrm{RAM}}(n)\log T^{\mathrm{RAM}}(n))$ and further using Lemma 2.11 we get $P^{\mathrm{dBSP}}(n) = O((S^{\mathrm{RAM}}(n))^2)$.

The first output is produced after the first instance travels through all processors, thus $T^{\mathrm{dBSP}}(n) = pqP(n) = O((T^{\mathrm{RAM}}(n))^{k+1}S^{\mathrm{RAM}}(n))$. Sum of all occupied registers over all processors, with addresses possibly increased by the factor of $\log q$ gives space complexity $S^{\mathrm{dBSP}}(n) = pqS^{\mathrm{RAM}}(n)\log q = O((T^{\mathrm{RAM}}(n))^{k+1}S^{\mathrm{RAM}}(n)) = O((T^{\mathrm{RAM}}(n))^{k+2})$. □

A *pipelined PRAM (pipe-PRAM)* is constructed from an ordinary PRAM by adding input and output arrays of registers basically in the same way as in Definition 3.21 (pipe-RAM) and Definition 3.22 (pipe-BSP). The pipe-PRAM[6] has the important property that the I/O registers are accessible by only the first $O(n^k)$ processors, for some constant $k > 0$. Each variant of the PRAM model defined in Section 2.3 can be transformed into a distinct version of the pipe-PRAM. Nevertheless, all these machines can simulate each other in polynomially equivalent time. A pipe-PRAM is said to be *uniform*, iff it starts cycling with the length of the cycle equal to period $P(n)$ on a sufficiently long sequence of identical inputs.

Definition 4.37 (Limited pipe-PRAM) *A pipe-PRAM with period $P(n)$ is called a limited pipe-PRAM iff it is uniform and its computation cycle contains a configuration C, such that there is a RAM algorithm working in space $S(n) = O(P^k(n))$ for some constant $k > 0$, which computes the contents of the i-th register of the j-th processor in configuration C, given the input of size n and numbers i, j.*

Lemma 4.38 *Let there exist a sequential RAM algorithm for problem $\mathcal{P}$ running in time $T^{\mathrm{RAM}}(n)$ and space $S^{\mathrm{RAM}}(n)$. Then the pipelined version of $\mathcal{P}$ can be solved by a limited pipe-PRAM with period $P^{\mathrm{PRAM}}(n) = O((S^{\mathrm{RAM}}(n))^2)$.*

[6] and pipe-BSP as well

Proof. We run the algorithm from Lemma 4.36 and use one register of the global memory per processor for communication. The communication, synchronization, and partitioning time is avoided, but the cost of send/recv instruction may increase by the factor of $\log p$. Only one instance is handled by a processor in any time ($q = 1$).

$$P^{\mathrm{PRAM}}(n) = \frac{T^{\mathrm{RAM}}(n)}{p} + S^{\mathrm{RAM}}(n) \log p + \log p .$$

Substitution of $p = T^{\mathrm{RAM}}(n)/S^{\mathrm{RAM}}(n)$ and $\log T^{\mathrm{PRAM}}(n) = O(S^{\mathrm{RAM}}(n))$ into the above formula produces $P^{\mathrm{PRAM}}(n) = O((S^{\mathrm{RAM}}(n))^2)$.

Clearly, the algorithm cycles with period $P(n)$. Every configuration consists of RAM configurations in different stages of processing and therefore can be generated by a sequential algorithm in space $S^{\mathrm{RAM}}(n)$. Hence the simulation can be performed by a limited pipe-PRAM. □

Lemma 4.39 *A limited pipe-PRAM computing with period $P(n)$ can be simulated by a RAM in space $S(n) = O(P^k(n))$.*

Proof. The RAM machine generates configuration C which exists in the cycle by definition. Then it simulates one period and checks that pipe-PRAM returns to C. The largest value manipulated — and thus the highest addressable register and the number of processors — during the period is bounded by $2^{O(P(n))}$, see Lemmas 2.15 and 2.16. Using the algorithms from the proofs of Theorems 2.19 and 2.21, a single period can be simulated by the SIMD CROW PRAM with period $O(P^3(n))$. Configuration C is gradually generated one register at a time, for each possible index of a processor and of a register. For every generated register value, the RAM (or TM) algorithm from the first part of the proof of Theorem 2.17 is executed to check whether the register will contain the same value after the single period. Checking runs recursively back in time from the last to the first step in the period. A call to `mem`$(0, i, j)$, stops the recursion and invokes the C generating algorithm to obtain the i-th register of j-th processor in time 0. The simulation needs space polynomial in the period of the SIMD CROW PRAM algorithm plus space polynomial in $P(n)$ needed to generate C. The total space used by the RAM computer is thus $S(n) = O(P^k(n))$. □

Definition 4.40 (Limited pipe-dBSP) *A pipelined dBSP machine computing with period $P^{\mathrm{dBSP}}(n)$ is limited, iff it can be simulated by a limited pipe-PRAM with period $P^{\mathrm{PRAM}}(n) = O((P^{\mathrm{dBSP}}(n))^j)$ for some constant $j > 0$.*

Theorem 4.41 *Limited pipe-PRAM and limited pipe-dBSP with $l(p) = O(p^k)$ for some $k > 0$ are members of class $\mathcal{C}_{\mathrm{weak}}$.*

Proof. Lemma 4.36 provides a pipelined dBSP algorithm for any problem with period polynomially equivalent to sequential space. A similar algorithm for the limited pipe-PRAM computer is given in Lemma 4.38. Its existence shows that also the dBSP computer is limited.

By definition, any limited pipe-dBSP machine can be simulated by a limited pipe-PRAM with period $P^{\mathrm{PRAM}}(n) = O((P^{\mathrm{dBSP}}(n))^j)$. This in turn is simulated by a RAM in space $S^{\mathrm{RAM}}(n) = O((P^{\mathrm{PRAM}}(n))^k) = O((P^{\mathrm{dBSP}}(n))^{jk})$ according to Lemma 4.39. □

The limited pipe-dBSP allows for space-efficient sequential simulation, because the startup phase of its computation has limited space complexity. The strictly pipelined dBSP achieves

the same goal by isolating individual instances. The following rather technical definition ensures that computation of an instance cannot utilize any information produced by other instances. Moreover, the amount of work which can be spent in an instance during a single period is limited.

Definition 4.42 *For a chosen input word w, a partitioning of PRAM registers into two sets is a partitioning into set $P_{w,t}$ of registers* pertinent *to w in step t and set $I_{q,t}$ of registers* independent *on w in step t, iff:*

1. $P_{w,t} \cap I_{w,t} = \emptyset$.
2. $P_{w,t} \cup I_{w,t}$ *contains all local and global registers.*
3. *Each register $r \in I_{w,t} \cap P_{w,t-1}$ contains value 0 in step t.*
4. *No instruction works with two registers r_1, r_2 such that $r_1 \in P_{w,t} \wedge r_2 \in I_{w,t}$.*
5. *If a processor works with registers from $P_{w,t}$ in step t and with registers from $I_{w,t}$ in step $t+1$, then its program counter contains 0 in step $t+1$, i.e., the processor executes the same instruction as in the beginning of the computation.*

Definition 4.43 (Strictly pipelined PRAM) *A pipelined PRAM computer with period $P(n)$ is strictly pipelined, iff it is uniform, the sets of registers pertinent to any pair of input words w_i and w_j, $a \neq b$, are disjoint at every step t, and no more work than $O(P^k(n))$ for some constant $k > 0$ is done on registers pertinent to a single input word during a period.*

Definition 4.44 (Strictly pipelined dBSP) *A pipelined dBSP machine computing with period $P^{\mathrm{dBSP}}(n)$ is strictly pipelined, iff it can be simulated by a strictly pipelined PRAM with period $P^{\mathrm{PRAM}}(n) = O((P^{\mathrm{dBSP}}(n))^j)$ for some constant $j > 0$.*

Lemma 4.45 *A strictly pipelined PRAM working with period $P(n)$ can be simulated by a RAM in space $S(n) = O(P^k(n))$.*

Proof. The strictly pipelined PTM starts cycling with period $P(n)$, because it is uniform. A new input word is read and computation of the new instance begins during the cycle. The new instance has no information about the other instances, thus it must enter the cycle to ensure cyclic behavior. The computation may use only $2^{O(P(n))}$ processors and registers due to the maximum operand of an instruction possible in time $P(n)$. Only limited amount of work, namely $O(P^k(n))$, can be done on the instance in one period. Such work allows for allocation of space up to $O(P^k(n))$ bits of memory. This memory must be made free for the next instance during the next period. Hence, total memory consumed by a single instance is bounded by $O(P^k(n))$.

The simulation algorithm takes the input and progressively executes all periods (cycles) with this input until the output is produced. A single period is simulated by the recursive backtrack algorithm from the first part of the proof of Theorem 2.17. The simulation space is polynomial in $P(n)$. The recursion is interrupted by a call to $\texttt{mem}(0, i, j)$, which reads the value of register i in processor j computed during the previous period. As no information from the registers pertinent to other instances can be utilized, only $O(P^k(n))$ registers pertinent to the single instance being processed have to be stored from one cycle to the next. Other registers can be assumed to contain 0. □

Theorem 4.46 *Strictly pipelined PRAM and strictly pipelined dBSP with $l(p) = O(p^k)$ for some $k > 0$ are members of class $\mathcal{C}_{\mathrm{weak}}$.*

Proof. The algorithm from Lemma 4.38 is a strictly pipelined PRAM algorithm. Consequently, the algorithm described in Lemma 4.36 is a strictly pipelined dBSP algorithm. Both algorithms run with period polynomially equivalent to the sequential space.

According to the definition, any strictly pipelined dBSP computer can be simulated by a strictly pipelined PRAM with period $P^{\mathrm{PRAM}}(n) = O((P^{\mathrm{dBSP}}(n))^j)$. Lemma 4.45 then provides a RAM algorithm with space complexity $S^{\mathrm{RAM}}(n) = O((P^{\mathrm{PRAM}}(n))^k) = O((P^{\mathrm{dBSP}}(n))^{jk})$. □

4.6 Decomposable BSP and the Real World

The (non-decomposable) BSP can overestimate the time complexity of an algorithm, because communication parameters g, h are set according to the most pessimistic case of possible h-relations. For example, in a 2-dimensional mesh network with p processors, a message sent to a neighbouring processor has to travel through only a single network link. Another message sent between processors on the opposite sides of the mesh must traverse about $\sqrt{p}$ links. The example shows that h-relations with equal value of h may take various time when realized by an interconnection network. If an upper bound on time of some BSP algorithm implemented on a real computer is to be found, BSP parameters g and l should have values corresponding to the worst case of possible communication requests. Many important algorithms do not need general communication abilities (at least at some parts of computation) and optimized message passing is then much faster than generic h-relations.

On the other hand, dBSP estimates are too optimistic when compared to a real parallel computer. For example, suppose that a computer has an interconnection network with the 2-dimensional mesh topology. Then both $g(p)$ and $l(p)$ are of order $\sqrt{p}$. In a cluster of $p' < p$ processors, we have $g(p)$ and $l(p)$ equal to $\sqrt{p'}$. But such speedup can be realized on the mesh network only if the clusters are submeshes. If, for example, each dBSP cluster maps into a column of the mesh (as in Algorithm 4.10), then the parameters should remain $\sqrt{p}$ and not $\sqrt{p'}$. Hence the real cost will be greater than predicted by the dBSP model. The real execution time will be between the lower bound given by dBSP and the upper bound obtained using BSP. The dBSP model can provide indication whether it is worthwhile to optimize a parallel BSP algorithm for a particular network topology. If the cost is smaller on the dBSP model than on the BSP, then a suitable allocation of processes to physical processors can yield execution time close to the dBSP prediction and thus much better performance than when using some random allocation (assumed by the BSP model). On the other hand, if both times are roughly equivalent, then the BSP algorithm could not probably be improved too much.

The dBSP model could be made more accurate by taking into account the topology of the underlying network. We could define distance $d(i, j)$ between every pair $\langle i, j \rangle$ of processors. Values of parameters g and l in cluster C would not depend on the size of C, but instead on the maximum interprocessor distance in C, i.e., on $\max_{i,j \in C} d(i, j)$. This model retains the advantageous feature of BSP that architectural details are hidden from the algorithm. The disadvantage is that clusters of the same size, but containing different processors, do not have the same values of communication parameters. Thus, the processors are not interchangeable.

The dBSP model supports locality expressed by partitioning processors into noninteracting clusters. This approach is directly applicable to some algorithms, e.g., broadcasting and matrix multiplication presented in Section 4.2. Simulation of cellular automata described in the same section does not allow any set of processors to be isolated. The locality property of a 1-dimensional CA limits the distance to which a particular cell can influence states of other cells. During t time steps, only cells not farther than t are influenced. Only direct neighbours communicate in a cellular automaton, therefore the natural partitioning would be into pairs. Unfortunately, this would require frequent repartitioning, because a half of necessary communication links would cross cluster boundaries. The Algorithm 4.13 is a compromise between fast communication in small clusters and infrequently required repartitioning in larger clusters. A better solution could be a dBSP computer equipped with more than one router (communication network). Every processor would be connected to all routers and the routers would be partitioned separately. Thus two processors belonging to different clusters of one router could still communicate using some other router. The main problem of such extension of the dBSP model is that two routers with constant sized clusters suffice to simulate any Boolean circuit. Hence a dBSP machine with any parameters g and l would be able — after some setup phase — to compute as fast as a second class device.

Chapter 5

Conclusion

The thesis has introduced a series of new results. Especially, we have proposed formal definitions of the BSP and dBSP models of parallel computers. We have analyzed their relations to other models of computation. We have presented a number of basic dBSP algorithms which achieve a significant speedup in comparison with their BSP counterparts. We have also stated more precisely some previously known results concerning the class of weak parallel machines and pipelined parallel Turing machines. The main conclusion is that computational power of the BSP and dBSP computers can be tuned in a broad range from the first to the second machine class. Communication locality of parallel algorithms can be exploited by the decomposable BSP model. Moreover, the dBSP machines — contrary to ordinary BSP — fit well to the concept of pipelined parallelism expressed by the class of weak parallel machines. Table 5.1 summarizes membership of various models in machine classes. Only cells containing results proved in previous chapters and filled, other are left empty (even if the respective facts are known).

The BSP model is usually studied as a tool for design and analysis of practical algorithms. In this thesis, it has been shown that a BSP computer is also a versatile model of parallel computation from the point of view of the computational complexity theory. The dBSP model adds some novel features important especially for pipelined computation. In Section 4.6, we have outlined further modifications of the model. Analysis of their features is a theme for further research.

Model	**Parameters**	$\in \mathcal{C}_1$	$\in \mathcal{C}_2$	$\in \mathcal{C}_{\text{weak}}$
Turing machine		yes (Def. 2.3)		
RAM		yes (Th. 2.12)		
pipe-RAM (Def. 3.21)				no (Cor. 3.24)
PRAM (Def. 2.13, 2.18, 2.20)			yes (Th. 2.21)	
limited PPTM (Def. 2.27)				yes (Th. 2.28)
strictly pipelined PPTM (Def. 2.30)				yes (Th. 2.31)
uniform PPTM (Def. 2.24)				no (Cor. 2.34)
BSP (Def. 3.2)	$p = \text{const.}$	yes (Th. 3.10)		
	$l(p) = \Omega(p^a)$	yes (Th. 3.13)		
	$g(p) = \Omega(p^a)$	yes (Th. 3.14)		
	$g(p) = O(\log^a p)$ $l(p) = O(\log^b p)$		yes (Th. 3.16)	
	$\log^k p < g(p) < p^a$ $\log^k p < l(p) < p^a$	no (Th. 3.16)	no (Th. 3.16)	
pipe-BSP (Def. 3.22)				no (Cor. 3.25)
work-preserving BSP (Def. 3.26)				no (Cor. 3.28)
dBSP (Def. 4.1)	$p = \text{const.}$	yes (Th. 4.30)		
	$l(p) = \Omega(p^a)$	yes (Th. 4.31)		
	$l(p) = O(\log^b p)$		yes (Th. 4.32)	
	$\log^k p < l(p) < p^a$	no (Th. 4.34)	no (Th. 4.34)	
pipe-dBSP (Sect. 4.5.4)				no (Cor. 4.35)
limited pipe-PRAM (Def. 4.37)				yes (Th. 4.41)
limited pipe-dBSP (Def. 4.40)	$l(p) = O(p^k)$			yes (Th. 4.41)
strictly pipelined PRAM (Def. 4.43)				yes (Th. 4.46)
strictly pipelined dBSP (Def. 4.44)	$l(p) = O(p^k)$			yes (Th. 4.46)

Table 5.1: Overview of membership in machine classes

Acknowledgements

I am grateful to Jiří Wiedermann for supervising my study for three years and also my preparation of this thesis. Numerous long and fruitful discussions with him helped me to get the necessary insight into the theory of models of computation and the related theory of computational complexity. I thank Peter van Emde Boas for his comments to my papers and especially his question whether the renumbering property of the original dBSP model could be beneficial in some cases (the positive answer is illustrated by Algorithm 4.23). Thanks to Bill McColl, Klaus-Jörn Lange, Rolf Niedermeier, and Alexandre Tiskin for discussing various aspects of parallel computing in general and the BSP model in particular.

My research and preparation of this thesis was partially supported by the GA ČR, grant No. 201/98/0717.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Massachusetts, 1974.

[2] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Proc. of European Symposium on Algorithms*, pages 17–30, 1995.

[3] A. Bäumker and W. Dittrich. Parallel algorithms for image processing: Practical algorithms with experiments. University of Paderborn, Germany.

[4] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster. The power of reconfiguration. *Journal of Parallel and Distributed Computing*, 13:139–153, 1991.

[5] M. Beran. Computational power of BSP computers. In *Proceedings of SOFSEM '98*, volume 1521 of *Lecture Notes in Computer Science*, pages 285–293. Springer-Verlag, 1998.
`http://www.ms.mff.cuni.cz/~beran/publications.html`

[6] M. Beran. Iterative local computation on the BSP model. *Neural Network World*, 8(4):441–461, 1998.

[7] M. Beran. Decomposable bulk synchronous parallel computers. In *Proceedings of SOFSEM '99*, volume 1725 of *Lecture Notes in Computer Science*, pages 349–359. Springer-Verlag, 1999.
`http://www.ms.mff.cuni.cz/~beran/publications.html`

[8] A. Bertoni, G. Mauri, and U. Sabadini. A characterization of the class of functions computable in polynomial time on ramdom access machines. In *Proceedings of STOC 13*, pages 168–176, 1981.

[9] G. Bilardi, K. T. Herley, A. Pietracaprina, G. Pucci, and P. G. Spirakis. BSP vs LogP. In *SPAA '96: Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 25–32. ACM Press, 1996.

[10] G. Bilardi and F. P. Preparata. Horizons of parallel computation. Technical Report CS-93-20, Department of Computer Science, Universita di Padova, 1993.

[11] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) library - design, implementation and performance. In *Proceedings of 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed*

Processing (IPPS/SPDP), San Juan, Puerto Rico, April 1999.
`http://www.uni-paderborn.de/~pub/`

[12] R. V. Book, S. A. Greibach, and B. Wegbreit. Time and tape bounded Turing acceptors and AFL's. *Journal of Computer and System Sciences*, 4:606–621, 1970.

[13] A. Borodin. On relating time and space to size and depth. *SIAM Journal on Computing*, 6:733–744, 1977.

[14] L. Boxer, R. Miller, and A. Rau-Chaplin. Some scalable parallel algorithms for geometric problems. Technical Report 96-12, SUNY at Buffalo Department of Computer Science, 1996.
`http://www.scs.carleton.ca/~bsp/literatur/B/B_Some.html`

[15] L. Boxer, Russ M., and A. Rau-Chaplin. Some scalable parallel algorithms for geometric problems (extended abstract). In *Proceedings of 8th International Conference on Parallel and Distributed Computing and Systems (PDCS '96)*, Chicago, October 1996.
`http://www.scs.carleton.ca/~bsp/literatur/B/B_Some_paper.html`

[16] BSP machine parameters (table of contents).
`http://www.bsp-worldwide.org/implmnts/oxtool/params.html`

[17] R. Calinescu. Conservative discrete-event simulations on bulk synchronous parallel architectures. Technical Report PRG-TR-16-95, Oxford University Computing Laboratory, Oxford, 1995.

[18] A. K. Chandra, D. C. Kozen, and L.J. Stockmayer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[19] B. S. Chlebus, A. Czumaj, L. Gąsieniec, M. Kowaluk, and W. Plandowski. Algorithms for the parallel alternating direction access machine. In *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science (MFCS'96)*, volume 1113 of *Lecture Notes in Computer Science*, pages 267–278. Springer-Verlag, September 1996.
`http://www.scs.carleton.ca/~bsp/literatur/B/B_Some_paper.html`

[20] S. A. Cook. Towards a complexity theory of synchronous parallel computation. *L'Enseignement Mathématique*, 27:99–124, 1981.

[21] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15:87–97, 1986.

[22] S. A. Cook and R. A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7:354–375, 1973.

[23] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. van Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[24] J. R. Davy and P. M. Dew. *Abstract Machine Models for Highly Parallel Computers.* Oxford University Press, 1995.

[25] R. Diestel. *Graph Theory.* Springer-Verlag, 2000.
`ftp://ftp.math.uni-hamburg.de/pub/unihh/math/books/diestel/Diestel.Graph.Theory.II.pdf`

[26] S. R. Donaldson, J. M. D. Hill, and D. B. Skillicorn. Predictable communication on unpredictable networks: Implementing BSP over TCP/IP. Technical Report PRG-TR-40-97, Oxford University Computing Laboratory, Oxford, 1997.

[27] Ian Foster. *Designing and Building Parallel Programs.* Addison-Wesley, 1995.

[28] M. Garzon. *Models of Massive Parallelism — Analysis of Cellular Automata and Neural Networks.* Springer-Verlag, 1995.

[29] A. V. Gerbessiotis and C. J. Siniolakis. Primitive operations on the BSP model. Technical Report PRG-TR-23-96, Oxford University Computing Laboratory, Oxford, October 1996.

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

[31] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared memory model serve as a bridging model for parallel computation? In *Theory of Computing Systems*, pages 327–359. Springer-Verlag, May/June 1999.

[32] L. M. Goldschlager. A universal interconnection pattern for parallel computers. *Journal of the ACM*, 29:1073–1086, 1982.

[33] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards efficiency and portability: Programming with the BSP model. In *SPAA '96: Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12. ACM Press, 1996.

[34] S. E. Hambrusch and A. A. Khokhar. C^3: An architecture-independent model for coarse-grained parallel machines.

[35] F. C. Hennie and R. E. Stearns. Two-way simulation of multi-tape Turing machines. *Journal of the ACM*, 13:533–546, 1966.

[36] F. Meyer auf der Heyde and C. Scheideler. Communication in parallel systems. In *Proc. of SOFSEM '96*, volume 1175 of *Lecture Notes in Computer Science*, pages 16–33. Springer-Verlag, 1996.

[37] J. M. D. Hill, W. F. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, and T. Tsantilas amd R. Bisseling. BSPlib: The BSP programming library, 1998. BSPlib reference manual with ANSI C examples.
`http://www.bsp-worldwide.org/implmnts/oxtool/`

[38] J. M. D. Hill and D. B. Skillicorn. Practical barrier synchronisation. Technical Report PRG-TR-16-96, Oxford University Computing Laboratory, Oxford, 1996.

[39] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[40] P. M. Lewis II, R. E. Stearns, and J. Hartmanis. Memory bounds for recognition of context-free and context-sensitive languages. In *Proc. of the 6th IEEE Symposium on Switching Theory and Logical Design*, pages 191–202, 1965.

[41] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[42] B. H. H. Juurlink and H. A. G. Wijshoff. Communication primitives for BSP computers. *Information Processing Letters*, 58:303–310, 1996.

[43] B. H. H. Juurlink and H. A. G. Wijshoff. The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model. In *Proceedings of Euro-Par'96 (vol. II)*, volume 1124 of *Lecture Notes in Computer Science.* Springer-Verlag, 1996.

[44] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 869–941. Elsevier Science Publishers, Amsterdam, 1990.

[45] M. Kaufmann, H. Schröder, and J. F. Sibeyn. Asymptotically optimal and practical routing on the reconfigurable mesh. *Parallel Processing Letters*, 5(1):81–95, 1995.

[46] K.-J. Lange. On the distributed realization of parallel algorithms. In *Proc. of SOFSEM '97*, volume 1338 of *Lecture Notes in Computer Science*, pages 37–52. Springer-Verlag, 1997.

[47] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: A survey and synthesis. In *Proc. of the 28th Hawaii International Conference on System Sciences (HICSS)*, volume 2, pages 61–70, January 1995.

[48] J. M. R. Martin and A. V. Tiskin. BSP modelling of two-tiered parallel architectures. In B. M. Cook, editor, *Concurrent Computing — Architectures, Languages and Techniques*, pages 47–55. IOS Press, 1999.

[49] W. F. McColl. General purpose parallel computing. In A. M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Coputation. Proc. 1991 ALCOM Spring School on Parallel Computation*, pages 337–391. Cambridge University Press, 1993.

[50] W. F. McColl. A BSP realisation of Strassen's algorithm, May 1995.

[51] W. F. McColl. Bulk synchronous parallel computing. In John R. Davy and Peter M. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*, pages 41–63. Oxford University Press, 1995.

[52] W. F. McColl. Scalable computing. *Lecture Notes in Computer Science*, 1000:46–61, 1995.

[53] W. F. McColl and A. Tiskin. Memory-efficient matrix multiplication in the BSP model. *Algorithmica*, 24:287–297, 1999.

[54] M. Minsky. *Computation, Finite and Infinite Machines.* Prentice Hall, 1972.

[55] R. Niedermeier. Recursively divisible problems. In *Proceedings of 7th ISAAC*, volume 1178 of *Lecture Notes in Computer Science*, pages 183–192, 1996.

[56] R. Niedermeier. *Towards Realistic and Simple Models of Parallel Computation.* PhD thesis, Fakultät für Informatik, Eberhard-Karls Universität Tübingen, Tübingen, 1996. `http://www-fs.informatik.uni-tuebingen.de/~niedermr/publications/di2.ps.Z`

[57] PACE: Performance prediction for parallel & distributed computers. `http://www.dcs.warwick.ac.uk/~hpsg/pace_top.htm`

[58] E. Papaefstathiou, D.J. Kerbyson, G.R. Nudd, T.J. Atherton, and J.S. Harper. An introduction to the layered characterisation for high performance systems. Technical Report 335, Department of Computer Science, University of Warwick, Coventry, December 1997.

[59] V. R. Pratt and L. J. Stockmeyer. A characterization of the power of vector machines. *Journal of Computer and System Sciences*, 12:198–221, 1976.

[60] V. Ramachandran. A general purpose shared-memory model for parallel computation. In *Proceedings of IMA Workshop on Parallel Algorithms*, Institute for Mathematics and Its Applications, Minneapolis, MN, September 1996. `http://www.cs.utexas.edu/users/vlr/papers/ima97.ps`

[61] W. J. Savitch. Relations between deterministic and nondeterministic tape complexities. *Journal of Computer and System Sciences*, 4:177–192, 1970.

[62] W. J. Savitch and M. J. Stimson. Time-bounded random access machines with parallel processing. *Journal of the ACM*, 26:103–118, 1979.

[63] J. F. Sibeyn. Overview of mesh results. Technical Report MPI-I-95-1-018, Max Planck Institut für Informatik, Saarbrücken, Germany, 1995.

[64] J. F. Sibeyn and M. Kaufmann. BSP-like external-memory computation. In *Proceedings of CIAC'97 — 3rd Italian Conference on Algorithms and complexity*, volume 1203 of *Lecture Notes in Computer Science*, pages 229–240. Springer-Verlag, 1997.

[65] C. J. Siniolakis. On the complexity of BSP sorting. Technical Report PRG-TR-09-96, Oxford University Computing Laboratory, 1996. `http://www.comlab.ox.ac.uk/oucl/users/constantinos.siniolakis/index.html`

[66] C. Slot and P. van Emde-Boas. On tape versus core; an application of space efficient perfect hash function to the invariance of space. In *Proceedings of STOC'84*, pages 391–400, Washington D.C., 1984.

[67] L. Snyder. Experimental validation of models of parallel computations. *Lecture Notes in Computer Science*, 1000:78–100, 1995.

[68] Q. Stout. Meshes with multiple buses. In *Proc. of the 27th Annual Symposium on Foundations of Computer Science*, pages 264–273, 1986.

[69] A. Tiskin. The bulk-synchronous parallel random access machine. *Theoretical Computer Science*, 196:109–130, 1998.

[70] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms.* PhD thesis, University of Oxford, Oxford, 1998.

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

[72] P. van Emde Boas. Machine models and simulations. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 1–66. Elsevier Science Publishers, Amsterdam, 1990.

[73] P. van Emde Boas. The second machine class, model of parallelism. In J. van Leeuwen, J. K. Lenstra, and A. H. G. Rinnooy Kan, editors, *Parallel Computers and Computations, CWI Syllabus*, volume 9, pages 133–161. Centre for Mathematics and Computer Science, Amsterdam, 1985.

[74] J. van Leeuwen and J. Wiedermann. Array processing machines: An abstract model. *BIT*, 27:25–43, 1987.

[75] P. Vitányi. Locality, communication, and interconnect length in multicomputers. *SIAM Journal on Computing*, 17:659–672, 1988.

[76] J. S. Vitter and R. A. Simons. New classes for parallel complexity: A study of unification and other complete problems for $\mathcal{P}$. *IEEE Transactions on Computers*, C-35(5):403–418, May 1986.

[77] J. Wiedermann. Normalizing and accelerating RAM computations and the problem of reasonable space measures. In *Proceedings of ICALP'90*, volume 443 of *Lecture Notes in Computer Science*, pages 125–138. Springer-Verlag, July 1990.

[78] J. Wiedermann. Parallel Turing machines. Technical Report RUU-CS-84-11, Department of Computer Science, Utrecht University, Utrecht, 1984.

[79] J. Wiedermann. Weak parallel machines: A new class of physically feasible parallel machine models. In I. M. Havel and V. Koubek, editors, *Mathematical Foundations of Computer Science 1992, 17th Int. Symposium (MFCS'92)*, volume 629 of *Lecture Notes in Computer Science*, pages 95–111, Berlin, 1992. Springer-Verlag.

[80] J. Wiedermann. Quo vadetis, parallel machine models. *Lecture Notes in Computer Science*, 1000:101–114, 1995.

[81] L. D. Wittie. Communication structures for larger networks of microcomputers. *IEEE Transactions on Computers*, C-30(4):264–273, April 1981.