A Computational Model for Parallel and Hierarchical Machines

Carlo Fantozzi

Typeset on **October 28, 2003**

Sommario

Il parallelismo è una metodologia molto diffusa per aumentare la potenza di calcolo disponibile e fronteggiare le richieste computazionali sempre crescenti delle odierne applicazioni. Negli ultimi trent'anni, la soluzione offerta dal parallelismo è stata ampiamente studiata ed applicata; nonostante i successi ottenuti, tuttavia, la programmazione parallela rimane una attività laboriosa. Le difficoltà nella programmazione sono aggravate dalla persistente mancanza di un modello di calcolo parallelo che riscuota unanime consenso: una spiegazione di questo fatto è che un tale modello deve soddisfare requisiti parzialmente contrastanti.

In anni recenti, la comunità scientifica ha riconosciuto la necessità di un "modello ponte" (*bridging model*), capace di contemperare tali contrastanti necessità. Nella presente tesi si avanza la candidatura a bridging model del D-BSP, una estensione del popolare modello BSP di Valiant pensata per catturare una porzione sostanziale della struttura gerarchica presente nella rete di interconnessione dei calcolatori paralleli. La prima parte di questa tesi presenta esempi dell'utilità della gerarchia di rete così come modellata dal D-BSP. In particolare, nel Capitolo 2 si fornisce una rassegna di risultati che mostrano come il D-BSP sia più efficace e flessibile del BSP nel modellare sistemi paralleli reali senza significative ricadute sulla semplicità d'uso; nel Capitolo 3, poi, si dimostra che la località di rete incorporata nel D-BSP può essere sfruttata per implementare efficientemente una astrazione di memoria condivisa. Negli ultimi due capitoli della tesi (Capitoli 4 e 5) si affronta una ulteriore e interessante applicazione del parallelismo strutturato fornito dal D-BSP, studiando le interazioni tra località di rete e località negli accessi alla memoria. Attraverso l'uso di simulazioni, si dimostra che tale parallelismo strutturato può essere completamente ed automaticamente trasformato in località di accesso. Come caso particolare, le simulazioni forniscono algoritmi sequenziali efficienti per memoria gerarchica a partire da algoritmi paralleli efficienti. Nei limiti della nostra conoscenza, questo è il primo studio che istituisce una relazione tra località di rete in algoritmi paralleli ed entrambe le forme di località di accesso (località *spaziale* e località *temporale*) in algoritmi sequenziali.

Abstract

Parallelism is a well-established solution to provide available computing power for the ever-increasing needs of current applications. In the last thirty years, the solution offered by parallelism has been thoroughly studied and applied; however, despite its success, parallel programming remains a hard task. Difficulties are worsened by the persisting lack of a successful model of parallel computation: a justification for this fact is that such a model must meet partly conflicting requirements.

In recent years, the scientific community has recognized the need of a *bridging model* to reach a reasonable tradeoff among such conflicting needs. In this thesis, we advance the candidacy as a bridging model of D-BSP, an extension of the popular BSP by Valiant aimed at capturing a substantial fraction of the hierarchical structure in the interconnection network of parallel machines. The first part of this thesis is devoted to providing examples of the usefulness of the network hierarchy as provided by D-BSP. In particular, in Chapter 2 we survey some existing results which show that D-BSP exhibits higher effectiveness and portability than BSP in modeling actual parallel architectures, without significantly affecting the ease of use; in Chapter 3, we demonstrate that the coarse network locality incorporated in the D-BSP model can be exploited to implement a shared memory abstraction in an efficient way. In the last two chapters of the thesis (Chapters 4 and 5) we address another interesting application of structured parallelism, as exposed by D-BSP, by studying the interplay between network locality and locality of reference. By resorting to cross-simulations, we prove that such structured parallelism can be fully and automatically translated into locality of reference. As an important special case, our cross-simulations are employed to obtain efficient hierarchy-conscious sequential algorithms from efficient parallel ones. To the best of our knowledge, ours is the first work that establishes a relation between the network locality embodied in parallel algorithms and both forms of locality of reference (namely, *temporal* and *spatial* locality) in sequential algorithms for general hierarchies.

"If all sorts of heavy work of this kind could be easily and quickly, as well as certainly, done, by merely selecting or punching a few Jacquard cards and turning a handle, not only much saving of labour would result, but much which is now out of human possibility would be brought within easy reach."

From the report of the committee appointed to consider the advisability and to estimate the expense of constructing Mr. Babbage's Analytical Machine

Acknowledgements

As I am writing this, the ACG Laboratory at the University of Padova is empty: there is nobody but me, in front of my trusty iMac, typing into LaTeX the last words of my thesis. However, I was not alone during my doctoral studies. In the silence of the lab, the faces come to my mind of all the people with which I shared three years of my life. My first thought goes to my advisor, Geppino Pucci, who is much more than a master to me, and, even on work matters, has always done more than duty commanded him to. It is almost impossible to think of Geppino without thinking of Andrea Pietracaprina as well. Working with Andrea and Geppino was an invaluable experience, since with them I learned what "doing research" means: indeed, I have never felt like a student while I was with them. I would also like to express my gratitude to Gianfranco Bilardi: a few words from him were always enough to wide my horizons. I wish to extend my grateful thought to Cinzia, Filippo, Mauro, and all the friends and people in the Department of Information Engineering: as I am going to leave the Department soon, I would like to say an affectionate goodbye to all of them. As for my family, I will write something in Italian, so that mom and dad can read it: grazie mamma, grazie babbo per avermi sempre sopportato in tutti questi anni nonostante i miei errori. Dopo quello che è successo, voi siete l'unica cosa che mi resta.

For my friends who have been with me in all these years, as well as for all the people I unintentionally forgot to mention, I would simply like to say: thank you.

Contents

Chapter 1

Introduction

The advances in microelectronics and related fields have made us accustomed to an exponential growth of resources available in computing machines, be such resources measured in terms of raw computing power, memory size, storage space or communication speed. Indeed, if we look some fifty years back, so much progress has been made that the question arises of whether there is a real need for further improvement. Not surprisingly, the answer is positive.

Problems exist which cannot be conquered with the amount of resources available today. For instance, some projected experiments in Particle Physics (such as those related to CERN's Large Hadron Collider) rely on the storage and analysis of multi-petabyte sets of data, which is a hardly feasible task with current technology. As a further example, many polynomial-time problems in the realms of Cosmology, Biochemistry and Climatology require an amount of computing power that is much greater than the one of current supercomputers, so that, even taking exponential growth into account, many decades will pass before reasonably-sized instances can be solved. Moreover, there is a widespread awareness of an incoming decrease in the growth rate experienced today for computing performance: a temporary slowdown may be due to contingent limits of technology, but, ultimately, physical barriers will be reached that cannot be overcome.

Rather than waiting for faster, future-generation machines, a solution that has proven its effectiveness in providing vast amounts of computing power is that of employing multiple copies of a resource in the form which is available *today*: the application of this idea to the processing unit gave birth to *parallel computing*. In the last thirty years, parallelism has been thoroughly studied from both a theoretical and a practical point of view; a proof of its success is offered by the fact that the most powerful supercomputers in the world are all massively parallel, featuring thousands of processing units. Unfortunately, theory and practice have also shown that parallelization is a hard task. There are many reasons for this fact, the most straightforward being that a problem may not be parallelizable, but, even if this is not the case, parallel programming is more difficult than sequential programming because there are many more factors that must be taken into account. More specifically, the difficulty resides in the need of coordinating the operations of the different processing units: data must typically be exchanged between the units, and asynchrony must be handled.

The increased number of degrees of freedom in parallel programming is made more perilous by the lack of a *computational model* capable of encapsulating them in a widely accepted framework. Indeed, despite extensive research activity in the field, no model has proven capable of taking a unifying role similar to the one of the Von Neumann/RAM machine [AHU74] in the realm of sequential computation. An intuitive explanation for this phenomenon is that a parallel model must meet partly conflicting requirements. Ideally, a computational model should provide [HL95, SA$^+$92] an abstract view of a class of machines while accurately reflecting costs and resources for *all* the machines in the class. The model should be simple and general enough to facilitate the design and analysis of algorithms; at the same time, the results of the analysis performed on the model should give good predictions of performance on the machines targeted by the model. This definition indicates that a computational model must be simple, easy to use, accurate and portable on different machines: clearly, it is difficult to meet all these requirements at once. For example, the PRAM model [FW78] offers powerful yet easy-to-use primitives, such as synchronous computation and a shared memory with unit-time access to every cell. Unfortunately, the architecture of real-world machines is quite different from the one that is implied by the model: as a consequence, PRAM gives inaccurate performance predictions, and this is particularly severe in a field where the need for performance is a primary requirement. For the same reasons connected to performance, even the leading role of the sequential RAM model has been questioned, and models such as the HMM [AACS87] have been proposed to account for the hierarchical structure of memory in present computers.

In recent years, the need has been recognized of a *bridging model* [Val90a] which is capable of mediating among the conflicting needs we have just described. Such a model should be accurate enough to yield fair predictions, but it should be detached from contingent hardware issues: attention must be focused on phenomena that are *intrinsic* in a parallel computation, i.e. phenomena that will be still present in the long term, when the advancement of technology will solve all but fundamental issues. According to this observation, which subset of real-world features must be included in the bridging model? We think that an issue that is present in current machines and that also looks fundamental enough for inclusion is the problem of *moving information* around. With "information" we indicate any item that is needed by a computation to advance towards completion: this definition includes items such as input data, partial results, synchronization or global clock signals. It must be remarked that not all information exchanges are equally "difficult": in fact, there is both empirical and theoretical evidence that it is "cheaper" to move information between "near" neighbors, or to move two pieces of information that are "near" to one another. This phenomenon, which will be formally introduced later in the thesis, goes under the name of *locality*.

In the interconnection network of parallel machines, locality cannot be ignored. Consider, for example, that any device connected to such networks always has a dedicated interconnection wire, hence the bandwidth "offered" by the network is proportional to the number of devices: each device expects to send data at full wire speed. As a matter of fact, the internal topology of the network is seldom capable of keeping the promise. For instance, in the well-known mesh topology it is possible to communicate with near neighbors at full speed, but global communication between any two halves of the network is slowed down by the surface/volume ratio: we can say that the network exhibits a *hierarchy* of levels with different speeds. A hierarchical structure is also offered by the memory subsystem, which typically includes multiple levels of caches, main memory and disks: the exploitation of locality is, once again, crucial. For these reasons, we advocate that some form of locality must be included, at least to a certain degree, in the bridging model.

In the first part of this thesis, we advance the candidacy as a bridging model of D-BSP [dK96], a variant of the distributed-memory BSP model proposed by Valiant [Val90a] in the early nineties. BSP and D-BSP adhere to the paradigm of *bulk-synchronous* computation, which meets wide acceptance and popularity: a bulk-synchronous program alternates computation and global synchronization phases, and messages sent during a computation phase are available at their destinations after the next synchronization phase. D-BSP extends BSP by allowing submachines to synchronize and operate independently from one another; the cost of communication is proportional to the size of the submachines, hence D-BSP can capture a substantial fraction of the hierarchical structure of the interconnection network in parallel computers. The results we report in Chapter 2 provide quantitative evidence that D-BSP exhibits higher effectiveness and portability than BSP in modeling actual parallel architectures, without significantly affecting ease of use. Moreover, we show that D-BSP is able to amend some deficiencies of the standard BSP, avoiding the resort to additional ad-hoc provisions that had motivated the definition of other variants of the model.

A very desirable feature of a distributed-memory model is the ability to support a shared memory abstraction efficiently; among the other benefits, this feature allows to port the vast body of PRAM algorithms [JáJ92] on the model at the cost of a small time penalty. The problem of shared memory implementation has been widely studied in the literature, with the most efficient solutions resorting to randomization to mitigate the impact of network congestion on performance. In Chapter 3, we show how efficiency can also be attained by exploiting the coarse network locality incorporated in the D-BSP model: to this aim, we present a general, *deterministic* scheme to implement a shared memory space on any distributed-memory machine which exhibits a clustered structure. More specifically, we develop a memory distribution strategy and an access protocol for D-BSP; the parameters of the model are instantiated to reflect the characteristics of n-processor machines whose network subsystem exhibits a recursive structure with bandwidth $O(n^{1-\alpha})$ and latency $O(n^{\beta})$. Our main result is summarized in the following theorem:

Theorem 1 *For any value m upper bounded by a polynomial in n and for any $k \geq 0$ there exists a scheme to access n memory cells out of a shared pool of size m in time*

$$O\left(2^k n^{\alpha + \frac{\alpha(1-\alpha)}{2^k(2-\alpha)}} + kn^{\beta}\right)$$

on a D-BSP with bandwidth $O(n^{1-\alpha})$, latency $O(n^{\beta})$ and $O(3^k m)$ aggregate memory size.

Our scheme achieves provably optimal slowdown with constant memory redundancy for those machines where delays due to latency dominate over those due to bandwidth limitations; for machines where this is not the case, we are able to achieve a slowdown which is a mere logarithmic factor away from the natural bandwidth-based lower bound, at the expense of a polylogarithmic blow-up in the total memory size. We remark that a randomized scheme can also take advantage of the network proximity modeled by D-BSP: as a proof of this fact, we give a simple, randomized strategy for shared memory access that exhibits optimal slowdown with high probability.

In the second half of the thesis, namely in Chapters 4 and 5, we address another interesting application of structured parallelism as modeled in D-BSP, by studying the interplay between network locality and locality in the access to memory. To understand the objective of our investigation, recall that both the interconnection network and the memory subsystem of modern supercomputers exhibit a hierarchical structure, hence they both reward the use of locality. Now, the network hierarchy has been extensively investigated, and established techniques exist to efficiently deal with it: a consequence of this research is a vast body of parallel algorithms that exploit network locality. *Locality of reference* (i.e., locality in the access to memory) has only recently started to attract the interest of the scientific community; even fewer studies have dealt with locality of reference and parallelism at once, although it is quite natural to wonder whether the two phenomena are governed by the same laws. Indeed, if this is the case, the aforementioned body of parallel techniques can be turned to the development of memory-efficient algorithms. The few studies in this direction which are available in the literature [DDH03, DDHM99, SK97] are promising, since they show how bulk-synchronous computations yield hierarchy-conscious algorithms; however, the scope of such works is limited to a 2-level (disk/RAM) hierarchy. In this thesis, we move a step forward by proving that the structured parallelism modeled by D-BSP can be fully and automatically translated into locality of reference on multi-level hierarchies: to do this, we present a uniform scheme to simulate any computation designed for v processors on a v'-processor configuration with $v' \leq v$ and the same overall memory size. In Chapter 4, we first concentrate on a limited framework, where, beyond network locality, only *temporal* locality is included in the sequential processor-memory pair through the aforementioned HMM model; technically, we describe accesses to memory and network through a single cost function $f(x)$. Under widely accepted and unrestrictive regularity conditions on $f(x)$ and on the parallel computation, we obtain the following result.

Theorem 2 *A T-time program for a v-processor D-BSP whose nodes are HMM machines can be simulated in time $\Theta\left(Tv/v'\right)$ on a v'-processor instance of the same model, for any $1 \leq v' \leq v$.*

Notice that this result can be regarded as an analogous of Brent's lemma [Bre74] for BSP-like parallel machines featuring a hierarchical structure for both memory and interconnection network. The simulation exhibits optimal slowdown for a wide class of computations. As an important special case ($v' = 1$), our simulation can be employed to obtain efficient hierarchy-conscious sequential algorithms from efficient *fine-grained* ones; indeed, in Chapter 4 we use this technique to obtain algorithms for matrix multiplication, DFT and sorting that are optimal on the sequential HMM model.

In Chapter 5, we make an effort to gain insight on the further interplay between parallelism and *spatial* locality of reference; recall that a computation is said to exhibit spatial locality if consecutive data in memory are involved in consecutive operations. In computational models, spatial locality is typically shaped in the form of a block transfer facility that allows to move a set of consecutive data at a reduced cost: for example, the sequential $f(x)$-BT model [ACS87] is an extension of HMM where an access to cell x takes time $f(x)$, but a block of l cells ending at address x can be copied in time $f(x) + l$. Our most significant result for spatial locality concerns fine-grained D-BSP programs, that is, programs where each processor features a constant-size memory, and can be summarized as follows.

Theorem 3 *A T-time, fine-grained program for D-BSP with logarithmic cost function can be simulated in time $O(nT)$ on $f(x)$-BT, for any $f(x) = O(x^\alpha)$, $0 \le \alpha < 1$.*

The proof of this result provides an effective tool to automatically obtain efficient BT algorithms from the large body of parallel algorithms available in the literature. Our findings reveal that efficiency in the BT model can be achieved for an entire family of cost functions, and starting from algorithms for a D-BSP machine that exhibits a coarse level of submachine locality (i.e., the machine features a high-bandwidth interconnection): this is in contrast with our simulation results on HMM, which, by Theorem 2, required a strict matching between the cost functions of the parallel and the sequential models. With a certain degree of approximation, we can state that the availability of block transfer "flattens" the memory hierarchy, that is, it makes an algorithm partly insensitive to the access cost function; however, some examples provide evidence that a certain level of network locality must nonetheless be retained in order to achieve efficient algorithms. To the best of our knowledge, ours is the first work that establishes a relation between the network locality embodied in parallel algorithms and the combined exploitation of both temporal and spatial locality of reference in sequential algorithms for general hierarchies.

Portions of this thesis are based on joint works with Gianfranco Bilardi, Andrea Pietracaprina, and Geppino Pucci; these works have been published in the open literature. The work described in Chapter 2 appeared in [BFPP01]. The work described in Chapter 3 appeared in [FPP01] as an extended abstract, and will be published in full in [FPP03]. The work described in Chapter 4 appeared in [FPP02]. Finally, the results contained in Chapter 5 have been submitted for publication in [FPP]. These research activities were supported in part by CNR and MIUR of Italy under projects *Algorithms for Large Data Sets: Science and Engineering* and *Algorithmics for the Internet and the Web*.

Chapter 2

The D-BSP Model

Widespread use of parallel computers requires the availability of a suitable model of computation which simultaneously exhibits *usability*, regarded as ease of algorithm design and analysis, *effectiveness*, so that efficiency of algorithms in the model translates into efficiency of execution on some given platform, and *portability*, which denotes the ability of achieving effectiveness with respect to a wide class of target platforms. These properties appear, to some extent, conflicting. For instance, effectiveness requires modeling a number of platform-specific aspects that affect performance (e.g., interconnection topology), at the expense of portability and usability. The formulation of a *bridging model* that balances among these clashing requirements has proved a difficult task, as demonstrated by the proliferation of models in the literature over the years. The most successful bridging model is, unquestionably, Valiant's BSP [Val90a]. A number of BSP variants, such as D-BSP [dK96] and BSP* [BDMadH98], have been formulated over the years with the aim of improving the effectiveness of the original model.

This chapter provides quantitative evidence of the higher effectiveness and portability exhibited by D-BSP with respect to BSP, and shows that the network locality explicitly captured in the model is able to amend some deficiencies of BSP without resorting to additional ad-hoc provisions. The results we present, which recently appeared in [BFPP01, BPP99, FPP01], lead us to regard D-BSP as one of the most promising candidates among BSP variants and, in general, among bandwidth-latency models, to strike a fair balance among the conflicting features sought in a bridging model of parallel computation.

The rest of the chapter is structured as follows. Section 2.1 is a survey on bandwidth-latency models, including BSP and its main variants. Section 2.2 formally defines the restricted version of D-BSP that is adopted in this thesis; network locality is modeled according to a regular recursive structure, which greatly improves usability. In Section 2.3, we describe the methodology based on cross-simulations proposed in [BPP99] to quantitatively assess the higher effectiveness of D-BSP with respect to BSP, relatively to the wide class of processor networks. Then, in Section 2.4 we show that, for certain relevant computations and prominent topologies, D-BSP exhibits a much higher effectiveness than the one guaranteed by the general result, so that the impact due to the loss of locality caused by ignoring the detailed network topology in the model becomes negligible. In the same section, we provide evidence that D-BSP can be as effective as E-BSP in dealing with unbalanced communication patterns.

2.1 Bandwidth-Latency Models

In the last decade, a number of bridging models have been proposed which abstract a parallel platform as a set of processors and a set of either local or shared memory banks (or both) communicating through some interconnection. In order to ensure usability and portability over a large class of platforms, these models do not provide detailed characteristics of the interconnection but, rather, summarize its communication capabilities by a few parameters that broadly capture bandwidth and latency properties. The literature on the subject is overwhelming: in what follows, we focus our attention on models that are relevant to this thesis. For a broader view on parallel models, the interested reader is referred to books [Lei92] and papers [Goo93, HL95, MMT95, Vit91] which are available in the literature. (Experimental studies have also been performed to establish which model [GLR$^+$96, JW98] or cost function [APM$^+$00, BP00, PPA$^+$98] is most effective, but it is difficult to compare these studies since each of them adopts a different setting and a different evaluation strategy.)

The most popular example of bandwidth-latency model is Valiant's BSP [Val90a].

A BSP (*Bulk Synchronous Parallel*) machine is a set of n processors with local memory, communicating through a router. Computations are sequences of *supersteps*: in a superstep, each processor (i) reads the messages received in the previous superstep; (ii) performs computation on locally available data; (iii) sends messages to other processors; and (iv) takes part in a global barrier synchronization. A superstep is charged a cost of $w + gh + \ell$, where w (resp., h) is the maximum number of operations performed (resp., messages exchanged) by any processor in the superstep, and g and ℓ are parameters respectively related to the router's bandwidth and latency.

Similar to BSP is the LogP model proposed by Culler et al. [CKP+96] which, however, lacks bulk synchronization and imposes a more constrained message-passing style aimed at keeping the load of the underlying communication network below a specified capacity limit. A quantitative comparison between the two models has been carried out in [BHPP00, BHP+96] showing a substantial equivalence between LogP and BSP as computational models for algorithm design guided by asymptotic analysis.

In recent years, a number of BSP variants have been formulated in the literature by incorporating additional provisions aimed at improving the model's effectiveness relative to actual platforms without significantly affecting its usability and portability. Among these variants, the following ones are particularly relevant for this thesis.

In Miller's extension to BSP [Mil94], the time for sending a message is a nonlinear function of the amount s of exchanged data. Specifically, the cost of sending a message of size s is $g(s)h + l$, where $g(x) = (n_{1/2}/x + 1)g_\infty$; the quantity g_∞ is the theoretical bandwidth of the interconnection network, as achieved with a message of infinite size, and $n_{1/2}$ is the message size for which the bandwidth is one half of the theoretical peak.

The *Deluxe BSP* [BGMZ97], or (d, x)-BSP, is one of the first parallel models that deals with the memory bottleneck, and it does so by increasing the number m of memory modules over the number n of processors so as to hide latency. The n memory banks which are directly connected to a processor can be accessed by their owner with constant slowdown, and by the other $n-1$ processors through a message exchange; the $(x-1)n$ banks which do not belong to a processor are directly attached to the interconnection network, and are accessible by sending messages to them. To quantify memory access time, two parameters are required: the minimum number d of cycles that must intervene between accesses to the same memory bank (*bank contention*), and the ratio $x \geq 1$ between the number of memory banks and the number of processors (*expansion factor*). The cost of a superstep is $w + gs + dr + \ell$, where s is the maximum number of remote memory requests made by a processor and r is the maximum number of remote requests handled by a bank. Deluxe BSP is proved to yield accurate predictions of performance on the Cray C90 and J90 systems.

In BSP* [BDMadH98], the cost of communication during a superstep is the maximum, among all processors, of the quantity

$$\max\Big\{l, g \max\Big\{\sum_{i=1}^{h'} \lceil s_i/B \rceil, \sum_{i=1}^{h''} \lceil s'_i/B \rceil\Big\}\Big\}, \tag{2.1}$$

where h' is the maximum number of messages sent by the processor, h'' is the maximum number of messages received by the processor, s_i is the size in bytes of the i-th sent message, s'_i is the size of the i-th received message and, finally, B is the critical block size. The parameter g is interpreted as "the ratio of the total throughput of the whole system in terms of basic computational operations to the throughput of the router in terms of messages of size B delivered"; more intuitively, BSP* favors messages whose size is at least B, thus modeling the fact that block transfers minimize start-up costs and improve the throughput of the communication subsystem. Cost function (2.1) is quite impractical, since it depends on the size of every single message that is sent in the superstep; as a consequence, in [BDP99] the cost function has been simplified to $g(s + hB) + \ell$, where s is the maximum *aggregate* size of the messages exchanged by a processor and h is defined as in the standard BSP model. This function improves usability while retaining a good degree of effectiveness: we will use it in Chapter 5.

The *Extended BSP* [JW96], or E-BSP, aims at predicting more accurately the cost of supersteps with *unbalanced* communication patterns, i.e. patterns where the average number h_{ave} of messages exchanged by a processor is lower than the peak number h. The cost function of this model depends on h, on the overall number m of messages that are actually sent, and on the *locality* L, defined as the maximum distance between source and destination of any message in a linear ordering of the processors. Note that m may be significantly lower than the theoretical number hn of messages involved in an h-relation (n is the number of processors); the ratio hn/m is a measure of the imbalance in the communication pattern.

2.2 The D-BSP Model

The D-BSP (Decomposable BSP[1]) [dK96] is another extension of Valiant's BSP [Val90a] which aims at incorporating into the model some degree of network locality through clustering. In its most general definition, D-BSP is regarded as a set of n processor/memory pairs communicating through a router; the processors can be aggregated according to a collection of clusters, where each cluster c is able to independently perform its own sequence of supersteps and is characterized by its own g and ℓ parameters, typically increasing with the size of the cluster. The key advantage of the model is that communication patterns where messages are confined within small clusters have small cost, like in realistic platforms and unlike in standard BSP. In fact, we will show quantitatively that this advantage translates into higher effectiveness and portability of D-BSP over BSP, at the expense of only a moderate burden on the algorithm designer.

In the same paper [dK96] which introduces D-BSP, the model is extended by proposing Y-BSP as a way to account for unbalanced communication patterns. Within a cluster c of the Y-BSP, the cost of performing an h-relation followed by a processor synchronization is $hb(c) + (m/|c|)g(c) + \ell(c)$, where m is the overall number of messages involved in the h-relation, $g(c)$ and $\ell(c)$ are the bandwidth and latency parameters for the cluster, and $b(c)$ is a start-up cost. In D-BSP, anyway, clustering already enables routing of unbalanced communication patterns with a good degree of efficiency, making it unnecessary to further extend the cost model in

[1]As acknowledged by the authors, the name *Decomposable BSP* was suggested by Rob Bisseling.

the direction followed by Y-BSP and E-BSP. As a whole, we think that D-BSP is an attractive candidate, among bandwidth-latency models, to strike a fair balance among usability, effectiveness and portability.

It must be remarked that, according to the general definition of D-BSP, the partition of processor into clusters can change dynamically: this feature makes it possible to incorporate the fine details of point-to-point network interconnections, but we think that it complicates the specification of the model without a good reason. In fact, it can be shown [Lei85] that, within polylogarithmic factors, a binary decomposition tree is already an optimal descriptor of any network topology occupying a given volume of space. In accordance with the above observation, in this thesis we focus our attention on a restricted version of the D-BSP model (referred to as *recursive D-BSP* in [dK96]) where the collection of submachines has the following regular structure. Let n be a power of two; for $0 \leq i \leq \log n$, the n processors[2] are partitioned into 2^i fixed, disjoint *i-clusters* $C_0^{(i)}, C_1^{(i)}, \cdots, C_{2^i-1}^{(i)}$ of $n/2^i$ processors each, where the processors of a cluster are able to communicate among themselves independently of the other clusters. The clusters form a hierarchical, binary decomposition tree of the D-BSP machine: specifically, $C_j^{\log n}$ contains only processor P_j, for $0 \leq j < n$, and $C_j^{(i)} = C_{2j}^{(i+1)} \cup C_{2j+1}^{(i+1)}$, for $0 \leq i < \log n$ and $0 \leq j < 2^i$.

A D-BSP computation consists of a sequence of labelled supersteps. In an *i-superstep*, $0 \leq i \leq \log n$, each processor executes internal computation on locally held data and sends messages exclusively to processors within its i-cluster. The superstep is terminated by a barrier, which synchronizes processors within each i-cluster independently. It is assumed that messages are of constant size, and that messages sent in one superstep are available at the destinations only at the beginning of the subsequent superstep. Let $\boldsymbol{g} = (g_0, g_1, \ldots, g_{\log n})$ and $\boldsymbol{\ell} = (\ell_0, \ell_1, \ldots, \ell_{\log n})$. If each processor performs at most w local operations, and the messages sent in the superstep form an *h-relation* (i.e., each processor is source or destination of at most h messages), then, the cost of the i-superstep is upper bounded by

$$w + hg_i + \ell_i, \text{ for } 0 \leq i \leq \log n.$$

In other words, an i-cluster in isolation behaves like a BSP of parameters g_i and ℓ_i;

[2]In this thesis, with $\log(\cdot)$ we denote the base 2 logarithm.

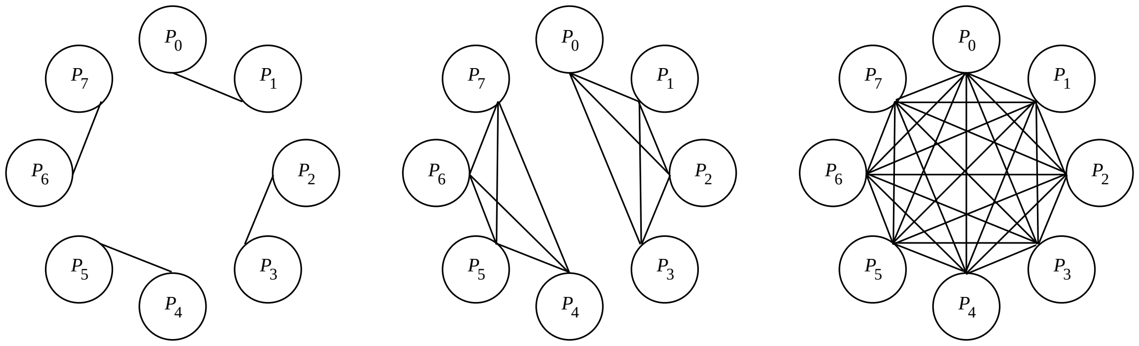

Figure 2.1: communication in a D-BSP$(n, \boldsymbol{g}, \boldsymbol{\ell})$ with $n = 8$ processors. The figure represents available communication paths in a 2-superstep (left), in a 1-superstep (middle) and in a 0-superstep (right).

the value of g_i and ℓ_i is related to the bandwidth and latency guaranteed by the router when communication occurs within i-clusters.

We refer to the model we have just described as a D-BSP$(n, \boldsymbol{g}, \boldsymbol{\ell})$. Figure 2.1 illustrates possible communications in a D-BSP$(n, \boldsymbol{g}, \boldsymbol{\ell})$ with 8 processors, and Figure 2.2 represents the execution on the same machine of a small program. Note that the standard BSP(n, g, ℓ) defined by Valiant can be regarded as a D-BSP$(n, \boldsymbol{g}, \boldsymbol{\ell})$ with $g_i = g$ and $\ell_i = \ell$ for every i, $0 \leq i \leq \log n$. In other words, D-BSP introduces the notion of proximity in BSP through clustering, and groups h-relations into specialized classes associated with different costs. This ensures full compatibility between the two models, which allows programs written according to one model to run on any machine supporting the other, the only difference being their estimated performance.

In this chapter, and in the following one, we will often exemplify our considerations by focusing on a class of D-BSP parameter values of particular significance. Namely, let α and β be two arbitrary constants, with $0 < \alpha, \beta < 1$. We will consider D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$ machines with

$$\begin{cases} g_i^{(\alpha)} = \Theta\left((n/2^i)^\alpha\right), \\ \ell_i^{(\beta)} = \Theta\left((n/2^i)^\beta\right), \end{cases} \text{for } 0 \leq i \leq \log n. \tag{2.2}$$

As we will see, these parameters capture a wide family of machines whose clusters are characterized by moderate bandwidth and moderate to high latency.

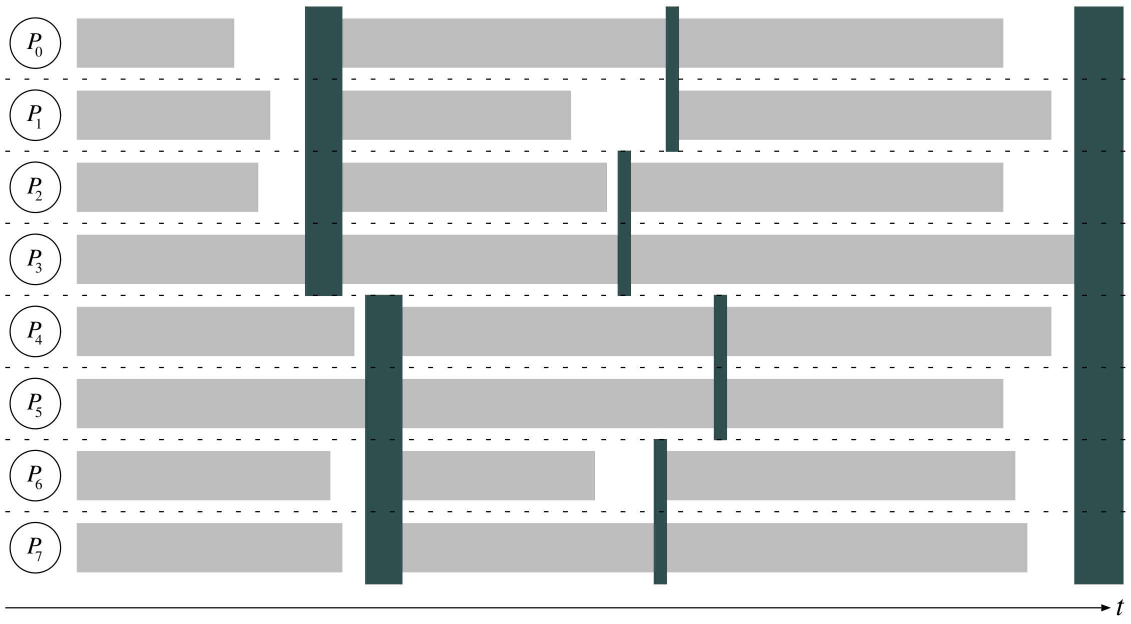

Figure 2.2: execution of a small program on a D-BSP$(n, \boldsymbol{g}, \boldsymbol{\ell})$ with $n = 8$ processors. The program includes a 1-superstep followed by a 2-superstep, and a final 0-superstep. Light grey boxes indicate computation, and dark grey boxes represent synchronization barriers.

2.3 D-BSP and Processor Networks

Intuitively, a computational model M, where designers develop and analyze algorithms, is *effective* with respect to a platform M', on which algorithms are eventually implemented and executed, if the algorithmic choices based on M turn out to be the right choices in relation to algorithm performance on M'. In other words, one hopes that the relative performance of any two algorithms developed on M reflects the relative performance of their implementations on M'. In order to attain generality, a computational model abstracts specific architectural details (e.g., network topology), while it incorporates powerful features that simplify its use but may not be exhibited by certain platforms. Therefore, in order to evaluate the effectiveness of a model M with respect to a platform M', we must establish the performance loss incurred by running algorithms developed for M (which do not exploit platform-specific characteristics) on M', and, on the other hand, we must assess how efficiently features offered by M to the algorithm designer, can be supported on M'.

More precisely, let us regard M and M' as two different machines, and let Π and Π' be the sets of optimal programs on M and M', respectively. We also define a *translation function* σ that associates a program $\mathcal{P} \in \Pi$ for a given computational problem to its counterpart $\mathcal{P}' \triangleq \sigma(\mathcal{P}) \in \Pi'$; by definition, the programs are optimal on the respective machines. In [BPP99] it is proposed to measure the effectiveness of M with respect to M' through the quantity

$$\eta(M, M') \triangleq \max_{\mathcal{P}_1, \mathcal{P}_2 \in \Pi} \frac{T(\mathcal{P}_1)}{T(\mathcal{P}_2)} \cdot \frac{T'(\sigma(\mathcal{P}_2))}{T'(\sigma(\mathcal{P}_1))},$$

where $T(\cdot)$ and $T'(\cdot)$ denote the execution time of a program on M and M', respectively. It is easy to see that $\eta(M, M') \geq 1$. We remark that η is an *inverse* measure of effectiveness: a value of $\eta(M, M')$ close to 1 implies that algorithms for the same problem exhibit similar running times on M and M' (a proof of high effectiveness), while a large value of $\eta(M, M')$ indicates that the performance on a program for M may not be preserved on M'.

An upper estimate of $\eta(M, M')$ can be obtained based on the ability of M and M' to simulate each other. Define $S(M, M')$ (resp., $S(M', M)$) as the minimum slowdown needed for simulating M on M' (resp., M' on M). In [BPP99] it is shown that the product

$$\delta(M, M') \triangleq S(M, M')S(M', M)$$

provides an upper bound to $\eta(M, M')$: as a consequence, effectiveness increases with decreasing $\delta(M, M')$ and is highest for $\delta(M, M') = 1$. When maximized over all platforms M' in a given class, this quantity provides an upper measure of the portability of M with respect to the class.

This approach based on simulations can be used to evaluate the effectiveness of D-BSP with respect to the class of processor networks. Let G be a connected n-processor network, where in one *step* each processor executes a constant number of local operations and may send/receive one point-to-point message to/from each neighboring processor (multi-port regimen). As is the case for all relevant network topologies, we assume that G has a decomposition tree $\{G_0^{(i)}, G_1^{(i)}, \cdots, G_{2^i-1}^{(i)} : \forall i,\ 0 \leq i \leq \log n\}$, where each $G_j^{(i)}$ is a connected subnet (*i-subnet*) with $n/2^i$ processors and $G_j^{(i)} = G_{2j}^{(i+1)} \cup G_{2j+1}^{(i+1)}$. By combining the routing results of [LMR99, LR99] one can easily show that for every $0 \leq i \leq \log n$ there exist suitable values g_i and ℓ_i related, respectively, to the bandwidth and diameter characteristics of the i-subnets, such that an h-relation followed by a barrier synchronization within an i-subnet can be implemented in $O(hg_i + \ell_i)$ time. Let M be a D-BSP$(n, \boldsymbol{g}, \boldsymbol{\ell})$ with these particular g_i and ℓ_i values, for $0 \leq i \leq \log n$. Clearly, we have that

$$S(M, G) = O(1). \tag{2.3}$$

Vice versa, an upper bound to the slowdown incurred in simulating G on M is provided by the following theorem proved in [BPP99, Theorem 3].

Theorem 2.3.1 *Suppose that at most b_i links connect every i-subnet to the rest of the network, for $1 \leq i \leq \log n$. Then, one step of G can be simulated on D-BSP$(n, \boldsymbol{g}, \boldsymbol{\ell})$ in time*

$$S(G, M) = O\left(\min_{n' \leq n}\left\{\frac{n}{n'} + \sum_{i=\log(n/n')}^{\log n - 1} (g_i \max\{h_i, h_{i+1}\} + \ell_i)\right\}\right), \tag{2.4}$$

where $h_i = \lceil b_{i-\log(n/n')}/(n/2^i)\rceil$, for $0 \leq i \leq \log n$.

Equations (2.3) and (2.4) can be applied to quantitatively estimate the effectiveness of D-BSP with respect to specific network topologies. Consider, for instance, the case of an n-node d-dimensional array. Fix $g_i = \ell_i = (n/2^i)^{1/d}$, for $0 \leq i \leq \log n$. Such D-BSP$(n, \boldsymbol{g}, \boldsymbol{\ell})$ can be simulated on G with constant slowdown. Since G has a decomposition tree with subnets $G_j^{(i)}$ that have $b_i = O\left((n/2^i)^{(d-1)/d}\right)$, the D-BSP simulation quoted in Theorem 2.3.1 yields a slowdown of $S(G, M) = O\left(n^{1/(d+1)}\right)$ per step. In conclusion, letting M be the D-BSP with the above choice of parameters, we have that $\delta(M, G) = O\left(n^{1/(d+1)}\right)$. The upper bound on $S(G, M)$ can be made exponentially smaller when each array processor has constant-size memory. In this case, by employing a more sophisticated simulation strategy [BPP99, Theorem 4] it is possible to prove that $S(G, M) = O\left(2^{\Theta(\sqrt{\log n})}\right)$, thus significantly improving D-BSP's effectiveness.

It is important to remark that the clustered structure of D-BSP provides a crucial contribution to the model's effectiveness. Indeed, it can be shown that if M' is a BSP(n, g, ℓ) and G is a d-dimensional array, then $\delta(M', G) = \Omega\left(n^{1/d}\right)$ independently of g, l and the size of the memory at each processor [BPP99, Theorems 1 and 2]. This implies that, under the δ metric, D-BSP is asymptotically more effective than BSP with respect to multidimensional arrays.

2.4 Effectiveness of D-BSP: Examples

We note that non-constant slowdown for simulating an *arbitrary* computation of a processor network on D-BSP is to be expected since D-BSP disregards the fine structure of the network topology, and, consequently, it is unable to fully exploit topological locality. However, for several prominent topologies and several relevant computational problems arising in practical applications, the impact of such a loss of locality is much less than what the above simulation results may suggest, and, in many cases, it is negligible.

Consider, for example, a processor network G whose bounded-degree topology has a recursive structure with flux $\Omega\left(n^{-\alpha}\right)$ and diameter $O\left(n^{\beta}\right)$, for arbitrary constants $0 < \alpha, \beta < 1$ (note that d-dimensional arrays satisfy these requirements). In this case, by combining the routing results of [LMR99, LR99] with those in [BPP99] it can be shown that a D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$ M can be supported on G with a slowdown $S(M,G)$ that is at most polylogarithmic[3]. Hence, algorithms devised on M can be implemented on G with at most polylogarithmic inefficiency. We now show that M allows us to develop algorithms for a number of relevant computational problems, which exhibit optimal performance when run on G. Hence, for these problems, the loss of locality of M with respect to G is negligible.

Broadcast and Prefix With *broadcast* we indicate a communication operation that delivers a constant-sized item, which is initially stored in a single processor, to every processor in the parallel machine. With the term *n-prefix* we indicate the calculation of a binary, associative operation $\oplus$ using the constant-sized elements of an array of length n; before the prefix starts processor P_j, $0 \leq j < n$, stores element

[3]Recall that the *flux* of a network $G = (V, E)$ is defined as the minimum among all subsets $U \subset V$, with $|U| \leq |V|/2$, of $|C(U, V - U)|/|U|$, where $C(U, V - U)$ denotes the edge-cut induced by U [LR99].

a_j of the array, and after the prefix it stores the quantity $a_0 \oplus a_1 \oplus \ldots \oplus a_j$. Note that n-prefix requires time $\Omega\left(n^\beta\right)$ to be performed on G [Lei92]. On D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$, we have:

Proposition 2.4.1 ([dK96]) *Any instance of single-item broadcast or n-prefix can be accomplished in optimal time $O\left(n^\alpha + n^\beta\right)$ on D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$.*

Proof For broadcast, we suppose that the item is initially stored in processor P_0: this hypothesis can be easily satisfied in time $O\left(n^\alpha + n^\beta\right)$ by means of a 0-superstep. The standard tree algorithm for broadcast consists of $\log v$ steps: during Step j, $0 \leq j < \log n$, processor P_i sends a copy of the item to processor P_{i+2^j} if it already owns the item, otherwise it does nothing. Correctness is proved by inductively showing that, at the beginning of Step j, processors $P_0, \ldots, P_{2^j-1}$ have a copy of the item. Since Step j requires a $(\log n - j - 1)$-superstep, the running time of the algorithm is

$$O\left(\sum_{j=0}^{\log n - 1} (2^{j+1})^\alpha + (2^{j+1})^\beta\right),$$

which evaluates to $O\left(n^\alpha + n^\beta\right)$.

For n-prefix, the solution we present is based on a recursive decomposition into two $n/2$-prefix subproblems, which are assigned to distinct submachines of size $n/2$. When invoked on a problem of size n, the algorithm behaves as follows.

1. If $n = 1$, then the algorithm returns the input unmodified.

2. If $n > 1$, then the algorithm is recursively invoked on the two $n/2$-prefix subproblems that are contained in clusters $C_0^{(1)}$ and $C_1^{(1)}$; these clusters behave as submachines with $n/2$ processors. When the subproblems have been solved, the prefix $a_{n/2-1}$ is broadcasted to the processors in cluster $C_1^{(1)}$, which update their current prefix by setting $a_j = a_{n/2-1} \oplus a_j$, for $n/2 \leq j < n$.

By using the broadcast algorithm described above, the time for n-prefix is given by the recurrence equation

$$T_{\text{PREFIX}}(n) = \begin{cases} 2T_{\text{PREFIX}}(n/2) + O\left(n^\alpha + n^\beta\right) & \text{for } n > 1, \\ O\left(1\right) & \text{for } n = 1. \end{cases}$$

The solution of the recurrence via the iteration method yields the stated running time of $O\left(n^{\alpha}+n^{\beta}\right)$. $\square$

Clearly, when $\alpha \leq \beta$ the implementation of the D-BSP algorithm on G exhibits optimal performance. We remark that optimality cannot be obtained using the standard BSP model. Indeed, results in [Goo96] imply that the direct implementation on G of any BSP algorithm for n-prefix, runs in time $\Omega\left(n^{\beta}\log n/\log(1+\lceil n^{\beta-\alpha}\rceil)\right)$, which is not optimal, for instance, when $\alpha = \beta$.

Sorting We call *k-sorting* a sorting problem in which k keys are initially assigned to each one of the n D-BSP processors and are to be redistributed so that the k smallest keys will be held by processor P_0, the next k smallest ones by processor P_1, and so on. It is easy to see that k-sorting requires time $\Omega\left(kn^{\alpha}+n^{\beta}\right)$ to be performed on G because of the bandwidth and diameter of the network. On D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$, we have:

Proposition 2.4.2 ([FPP01]) *Any instance of k-sorting, with k upper bounded by a polynomial in n, can be executed in optimal time $T_{\mathrm{SORT}}(k,n) = O\left(kn^{\alpha}+n^{\beta}\right)$ on D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$.*

Proof The algorithm we propose prescribes to sort the k keys inside each processor sequentially, then simulate bitonic sorting on a hypercube [Lei92] using the *merge-split* rather than the compare-swap operator. To complete a merge-split phase, two processors P_a, P_b perform the following operations.

1. Processor P_b sends its k keys to P_a in a single superstep.
2. P_a merges the keys coming from P_b with its own keys. Since the keys of both P_a and P_b are sorted, this phase takes time $\Theta(k)$.
3. P_a splits the new, sorted list in two, then it sends half of the list to P_b in a single superstep.

Processors adjacent along the i-th dimension of the cube are mapped to D-BSP processors within the same $(\log n - i - 1)$-cluster, for $0 \leq i < \log n$: as a consequence, the time to complete a merge-split phase during the simulation of the i-th dimension of the hypercube is $\Theta(k + kg_{i+1} + l_{i+1})$. The preliminary sorting is executed only once and it clearly takes time $\Theta(k \log k)$. To sum up, the overall running time for the k-sorting algorithm is

$$O\left(k \log k + \sum_{i=0}^{\log n - 1} (i+1)(kg_i + l_i)\right),$$

which simplifies to $O\left(kn^\alpha + n^\beta\right)$ if k is upper bounded by a polynomial in n. A simple bandwidth argument is sufficient to prove optimality. □

The algorithm of Proposition 2.4.2 is clearly optimal when ported to G. Again, a similar result is not possible with standard BSP: the direct implementation on G of any BSP algorithm for k-sorting runs in time $\Omega\left((\log n / \log k)(kn^\alpha + n^\beta)\right)$, which is not optimal for small k.

Routing We call (k_1, k_2)-*routing* a routing problem where each processor is the source of at most k_1 packets and the destination of at most k_2 packets. Observe that any (k_1, k_2)-routing is a $\max\{k_1, k_2\}$-relation, hence the "greedy" routing strategy where all packets are delivered within the same superstep requires, in the worst case, $\max\{k_1, k_2\} \cdot n^\alpha + n^\beta$ time on D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$, which is the best one could do on a BSP with matching bandwidth/latency parameters. However, a careful exploitation of the submachine locality exhibited by the D-BSP yields a better algorithm for (k_1, k_2)-routing.

Proposition 2.4.3 ([FPP01]) *Any instance of* (k_1, k_2)*-routing can be executed on D-BSP*$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$ *in optimal time*

$$T_{\text{rout}}(k_1, k_2, n) = O\left(k_{\min}^\alpha k_{\max}^{1-\alpha} n^\alpha + n^\beta\right),$$

where $k_{\min} = \min\{k_1, k_2\}$ *and* $k_{\max} = \max\{k_1, k_2\}$.

Proof We accomplish (k_1, k_2)-routing on D-BSP in two phases as follows.

1. For $i = \log n - 1$ down to 0, in parallel within each $C_j^{(i)}$, $0 \le j < 2^i$: evenly redistribute messages with origins in $C_j^{(i)}$ among the processors of the cluster.

2. For $i = 0$ to $\log n - 1$, in parallel within each $C_j^{(i)}$, $0 \le j < 2^i$: send the messages destined to $C_{2j}^{(i+1)}$ to such cluster, so that they are evenly distributed among the processors of the cluster. Do the same for messages destined to $C_{2j+1}^{(i+1)}$.

Note that the above algorithm does not require that the values of k_1 and k_2 be known *a priori*. It is easy to see that at the end of iteration i of the first phase each processor holds at most $a_i = min\{k_1, k_2 2^i\}$ messages, while at the end of iteration i of the second phase, each message is in its destination $(i+1)$-cluster, and each processor holds at most $d_i = \min\{k_2, k_1 2^{i+1}\}$ messages. Note also that iteration i of the first (resp., second) phase can be implemented through a constant number of prefix operations and one routing of an a_i-relation (resp., d_i-relation) within i-clusters. Putting it all together, the running time of the above algorithm on D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$ is

$$O\left(\sum_{i=0}^{\log n-1}\left(\max\{a_i, d_i\}\left(\frac{n}{2^i}\right)^\alpha + \left(\frac{n}{2^i}\right)^\beta\right)\right).$$

The theorem follows by plugging in the above formula the bounds for a_i and d_i derived above.

Optimality is, again, based on a bandwidth argument. If $k_1 \le k_2$, consider the case in which each of the n processors has exactly k_1 packets to send; the destinations of the packets can be easily arranged so that at least one 0-superstep is required for their delivery. Moreover, suppose that all the packets are sent to a cluster of minimal size, i.e. a cluster containing $2^{\lceil \log(k_1 n/k_2)\rceil}$ processors: the time to make the packets enter the cluster is

$$\Omega\left(k_2\left(\frac{k_1}{k_2}n\right)^\alpha + \left(\frac{k_1}{k_2}n\right)^\beta\right) = \Omega\left(k_1^\alpha k_2^{1-\alpha} n^\alpha\right). \tag{2.5}$$

The lower bound is obtained by summing up the quantity (2.5) and the time required by the 0-superstep. The lower bound for the case $k_1 > k_2$ is obtained in a similar way: this time each processors receives exactly k_2 packets, which come from a cluster of minimal size $2^{\lceil \log(k_2 n/k_1)\rceil}$. □

Standard lower bound arguments [SK94] show that this routing time is optimal for G.

As a corollary of Proposition 2.4.3, we can show that, unlike the standard BSP model, D-BSP is also able to handle unbalanced communication patterns efficiently, which was the main objective that motivated the introduction of E-BSP [JW96]. Let an (h, m)-relation be a routing instance where each processor sends/receives at most h messages, and a total of m messages are exchanged. Although a greedy routing strategy for an (h, m)-relation requires time $\Theta\left(hn^\alpha + n^\beta\right)$ on both D-BSP and BSP, the exploitation of submachine locality in D-BSP allows us to route any (h, m)-relation in time $O\left(\lceil m/n\rceil^\alpha h^{1-\alpha} n^\alpha + n^\beta\right)$, which is equal or smaller than the greedy routing time and it is optimal for G. Consequently, D-BSP can be as effective in dealing with unbalanced communication as E-BSP, where the treatment of unbalanced communication is a primitive of the model.

Chapter 3

Simulation of Shared Memory

Providing a shared address space on a distributed-memory parallel system is a fundamental problem which has been extensively investigated over the last two decades from both a theoretical and a practical perspective. In the theoretical setting, the problem has been regarded as the simulation of the PRAM model of parallel computation [FW78] over processor networks [JáJ92]. More precisely, PRAM simulation requires the development of a *scheme* to represent m shared objects (called *variables*) onto a network of $n \leq m$ processor/memory pairs in such a way that any n-tuple of variables can be read/written efficiently by the processors. The time required by a parallel access to an arbitrary n-tuple of variables is referred to as the *slowdown* of the scheme. In this chapter, we present both randomized and deterministic strategies to endow D-BSP with an efficient shared memory abstraction. Our results indicate that the network locality modeled by D-BSP can be exploited to reduce significantly the impact of network congestion and bank contention during an access to the shared address space.

The rest of the chapter is organized as follows. Section 3.1 summarizes existing results concerning the simulation of shared memory on processor networks. Section 3.2 sketches our original contributions on the subject. Section 3.3 focuses on the memory organization employed by our deterministic scheme, and on the graphs involved; section 3.4 describes the two main components of the simulation algorithm, namely the copy selection step (Subsection 3.4.1) and the protocol to access the selected copies (Subsection 3.4.2). Section 3.5 briefly discusses issues related to the constructivity of the deterministic scheme. Finally, Section 3.6 describes a simple, randomized scheme for shared memory access that exhibits optimal slowdown with high probability.

3.1 Previous Work

In the literature, the PRAM simulation problem has been solved with both randomized and deterministic schemes. Randomized schemes typically distribute the variables among the memory modules local to the processors through one (or more) hash functions, chosen from a suitable universal class. The properties of these functions guarantee, with high probability, that the variables be evenly distributed among the modules so that an access to any n-tuple of variables incurs low congestion both at the memory modules and across the network. Randomized schemes based on this strategy have been proposed to simulate a PRAM step, with high probability, in $O(\log\log\log n \log^* n)$ time on the complete network [CMadHS00], in $O(\log n)$ time on the butterfly [Ran91] and in $O\left(dn^{1/d}\right)$ time on the d-dimensional mesh [LMRR94]. Work-optimal randomized schemes are presented in [Val90b], to simulate an $(n \log n)$-processor PRAM on an n-processor hypercube with slowdown $O(\log n)$, and in [GMR99], to simulate an $(n \log\log n)$-processor PRAM on an n-processor Optical Communication Parallel Computer with slowdown $O(\log\log n)$.

In contrast, deterministic schemes require a redundant representation of the address space where every variable is replicated into ρ copies distributed among the memory modules through a map which exhibits suitable expansion properties. Such expansion properties are needed to avoid trivial worst cases, where all of the copies of some n-tuple of variables are confined within few memory modules, or, more generally, within a low-bandwidth region of the network. The parameter ρ is referred to as the *redundancy* of the scheme. The main idea, originally introduced in [MV84] and subsequently refined in [UW87], is that any access (read or write) to a variable is satisfied by reaching only a subset of its copies which is suitably chosen to reduce congestion while ensuring consistency (i.e., a read access must always return the most updated value of the variable). Based on the above idea, deterministic schemes have been devised in the literature for a number of network topologies. In particular, for m polynomial in n, schemes are known to simulate a PRAM step on a complete network in time $O(\log n)$ [AHMP87], and on a mesh of trees with n processors and n^2 switching elements in time $O\left(\log^2 n/\log\log n\right)$ [LPP90]. For arbitrary values of m, schemes achieving polylogarithmic worst-case slowdown have been proposed for an expander-based network [HB94], and for an augmented mesh of trees [Her96]. By employing suitable splitting/combining techniques, the deterministic scheme presented in [HPP01] attains $O\left(\sqrt{n\log(m/n)}\right)$ slowdown on the pruned butterfly (a variant of the fat tree), and $O\left(n^{1/d}(\log(m/n))^{1-1/d}\right)$ slowdown on a d-dimensional mesh. In the same work, a general argument is provided to bound from below the worst-case slowdown of any deterministic simulation scheme as a function of certain bandwidth characteristics of the interconnection.

All of the aforementioned deterministic schemes crucially exploit the specific structure of the underlying network topology and, consequently, are not easily applicable to other topologies. Moreover, they require redundancy logarithmic in m and rely on the existence of very powerful expanding graphs of nonconstant degree for which no explicit construction is known.

The first deterministic schemes that are also fully constructive have been devised in [PP97] for the complete network. For $m = \Theta\left(n^{3/2}\right)$, $m = \Theta\left(n^2\right)$ and $m = \Theta\left(n^3\right)$, these schemes attain worst-case slowdown of $O\left(n^{1/3}\right)$, $O\left(n^{1/2}\right)$ and $O\left(n^{2/3}\right)$, respectively, while using constant redundancy. In a recent work [PPS00], a deterministic scheme for an n-processor mesh has been presented, which achieves $O\left(\sqrt{n}\log n\right)$ slowdown and features a novel memory organization based on weakly expanding graphs that can be explicitly constructed for a range of memory sizes. Moreover, the memory map exhibits a hierarchical structure tailored to the natural recursive decomposition of the mesh into submeshes and shows promise to be portable to other clustered architectures.

3.2 Original Contributions

Building on the techniques of [PPS00], in this chapter we develop a general scheme to implement shared memory on parallel machines with a clustered structure, achieving a worst-case slowdown which is optimal, or close to optimal, with respect to the limitations imposed by the machine's bandwidth/latency characteristics. The scheme is designed for the D-BSP model, which, as we showed in Chapter 2, can be efficiently supported on a wide class of clustered architectures by a suitable setting of the model's parameters: the choice of D-BSP, therefore, allows us to achieve generality. Our main result is summarized in the following theorem.

Theorem 3.2.1 *For any value m upper bounded by a polynomial in n and for any $k \geq 0$ there exists a scheme to implement a shared memory of size m on D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$ with redundancy $\Theta\left(3^k\right)$ and slowdown*

$$O\left(2^k n^{\alpha + \frac{\alpha(1-\alpha)}{2^k(2-\alpha)}} + kn^\beta\right).$$

The scheme requires only weakly expanding graphs of constant degree and can be made fully constructive for $m = O\left(n^{3/2}\right)$ and $\alpha \geq 1/2$.

(Recall that D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$ was defined in Section 2.2 as an n-processor D-BSP where the bandwidth and latency parameters for a cluster C are set to $|C|^\alpha$ and $|C|^\beta$, respectively.) The following corollary is an immediate consequence of the theorem.

Corollary 3.2.2 *For any value m upper bounded by a polynomial in n there exists a scheme to implement a shared memory of size m on D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$ with optimal slowdown $O\left(n^\beta\right)$ and constant redundancy, when $\alpha < \beta$, and slowdown $O\left(n^\alpha \log n\right)$ and redundancy $O\left(\log^{1.59} n\right)$, when $\alpha \geq \beta$. The scheme requires only weakly expanding graphs of constant degree and can be made fully constructive for $m = O\left(n^{3/2}\right)$ and $\alpha \geq 1/2$.*

The importance of the above results is twofold. First, the proposed scheme is general and can be implemented on any machine supporting the D-BSP abstraction. Second, Corollary 3.2.2 shows, for the first time in the literature, that optimal worst-case slowdowns for shared memory access are achievable with constant redundancy for machines where latency overheads dominate over those due to bandwidth limitations, as is often the case in network-based parallel machines. On the other hand, the lower bound proved in [HPP01] shows that under reasonable assumptions, the performance of our scheme is not far from optimal for bandwidth-limited machines.

Indeed, consider a machine that supports D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$ with $\alpha \geq \beta$, and assume that each subset of processors associated with an i-cluster is connected to the rest of the network by $O\left((n/2^i)^{1-\alpha}\right)$ links (note that such a machine cannot support a D-BSP$(n, \boldsymbol{g}^{(\alpha')}, \boldsymbol{\ell}^{(\beta)})$ with $\alpha' < \alpha$). The result in [HPP01, Theorem 9] implies that the minimal slowdown achievable by any scheme like ours which dispatches an individual message for each copy to be accessed is $\Omega\left(n^{\alpha}(\log(m/n)/\log\log(m/n))^{1-\alpha}\right)$, and that in order to achieve such slowdown redundancy $\Omega\left(\log(m/n)/\log\log(m/n)\right)$ is necessary.

Note also that, unlike the other classical deterministic schemes in the literature, the above scheme solely relies on expander graphs of mild expansion, hence it can be made fully constructive for a significant range of the parameters involved. Such mild expanders, however, are only able to guarantee that the copies of an arbitrary n-tuple of variables be spread among $O\left(n^{1-\epsilon}\right)$ memory modules, for some constant $\epsilon < 1$. Hence the congestion at a single memory module can be as high as $O\left(n^{\epsilon}\right)$ and the clustered structure of D-BSP is essential in order to achieve good slowdown. In fact, any deterministic strategy employing these graphs on BSP(n, g, ℓ) could not achieve better than $\Theta\left(gn^{\epsilon}\right)$ slowdown.

As mentioned before, our scheme builds upon the one presented in [PPS00], whose design exploits the recursive decomposition of the underlying interconnection to provide a hierarchical, redundant representation of the shared memory based on $k+1$ levels of logical modules, an organization which fits well with the hierarchical nature of D-BSP. More specifically, each variable is replicated into $r = O(1)$ copies, and the copies are assigned to r logical modules of level 0. In turn, the logical modules at the i-th level, $0 \leq i < k$ are replicated into three copies, which are assigned to three modules of level $i+1$. This process eventually creates $r3^k = \Theta\left(3^k\right)$ copies of each variable, and 3^{k-i} replicas of each module at level i. The number (resp., size) of the logical modules decreases (resp., increases) with the level number, and their replicas are assigned for storing to distinct subnetworks of appropriate size.

The key ingredients of the above memory organization are represented by the bipartite graph that governs the distribution of the copies of the variables among the modules of the first level, and by the graphs which govern the distribution of

Problem	**Execution Time**
Broadcast	$O\left(n^\alpha + n^\beta\right)$
Prefix computation	$O\left(n^\alpha + n^\beta\right)$
k-sorting	$O\left(kn^\alpha + n^\beta\right)$ if $k = \text{poly}(n)$
(k_1, k_2)-routing	$O\left(k_{\min}^{\alpha} k_{\max}^{1-\alpha} n^\alpha + n^\beta\right)$, where $k_{\min} = \min\{k_1, k_2\}$ and $k_{\max} = \max\{k_1, k_2\}$

Table 3.1: execution times on D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$ of some common primitives.

the replicas of the modules at the subsequent levels. The former graph is required to exhibit some weak expansion property and its existence can always be proved through combinatorial arguments although, for certain memory sizes, explicit constructions can be given. In contrast, all the other graphs employed in the scheme require expansion properties that can be obtained by suitable modifications of the BIBD graph [HJ86], and can always be explicitly constructed. In fact, the choice of these latter graphs is where the memory organization adopted here mainly differs from the one presented in [PPS00]. In particular, unlike [PPS00], these graphs are not simply subgraphs of the BIBD, but their construction requires some nontrivial rearrangements to make them suitable for the clustered structure of D-BSP while maintaining good expansion properties.

For an n-tuple of variables to be read/written, the selection of the copies to be accessed and the subsequent access to the selected copies are performed via suitable protocols, similar to the ones in [PPS00], which can be implemented through a combination of prefix, sorting and routing primitives: to obtain the results stated above, we employ efficient D-BSP implementations for these primitives, including an optimal implementation of (k_1, k_2)-routing. Such primitives were extensively analyzed in Section 2.4; for ease of reference, the results of the analysis are summarized in Table 3.1.

3.3 Memory Organization

The Hierarchical Memory Organization Scheme (HMOS) that governs the distribution of the copies of the m shared variables among the local memories of the n

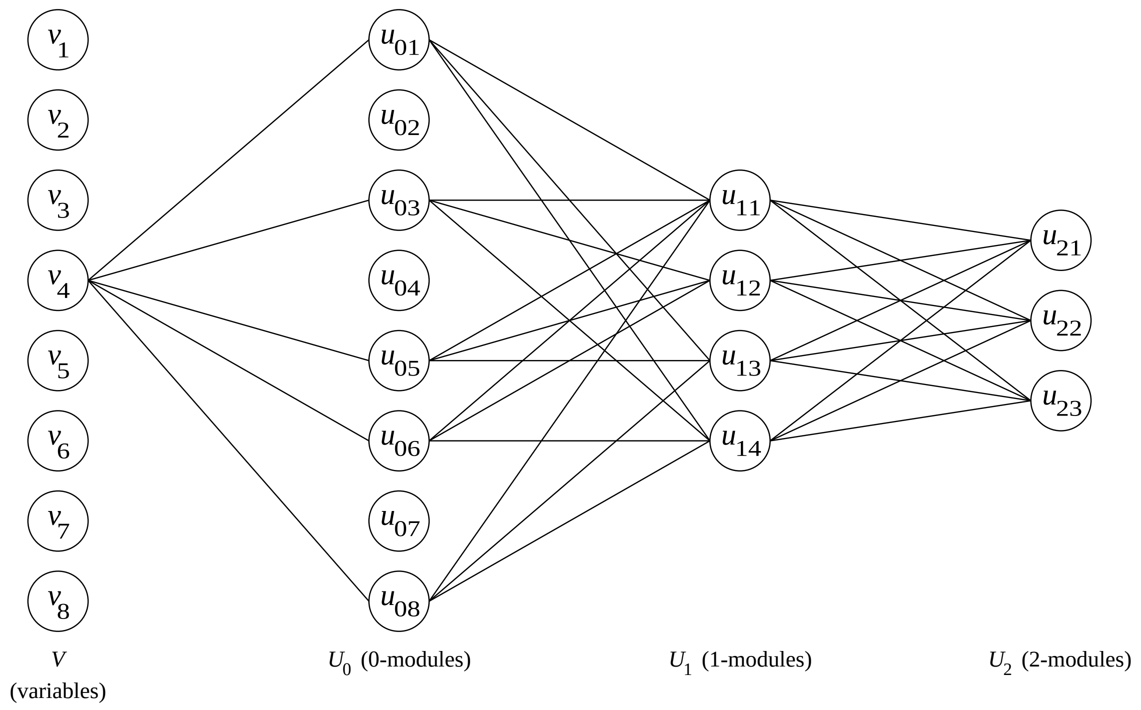

Figure 3.1: a sample HMOS built upon $|V| = m = 8$ variables, with $k = 2$ levels of i-modules, $|U_0| = 8$ 0-modules, $|U_1| = 4$ 1-modules, $|U_2| = 3$ 2-modules and $r = 5$. To reduce the size of the HMOS, thus making it easier to read, the figure does not respect all the constraints on the parameters that are listed in Section 3.3. For the same reason, only the edges associated with the copies of variable v_4 are shown.

processors is a cascade of bipartite graphs obtained as a modification of the one introduced in [PPS00]. A sample HMOS is depicted in Figure 3.1. In what follows we recall how the HMOS is structured and point out the differences with the version in [PPS00].

Let V denote the set of variables, and let U_i, $0 \leq i \leq k$, denote a set of nodes referred to as *i-modules*, where $k = O(\log \log n)$ is a suitable nonnegative integer that will be specified in the analysis. The U_i's can be regarded as nested collections of variables, and are obtained as follows. First, each variable is replicated into $r = O(1)$ copies, r odd, which are assigned to distinct 0-modules. The contents of each 0-module, viewed as an indivisible unit, are in turn replicated into 3 copies, which are assigned to distinct 1-modules. In general, the contents of each $(i-1)$-

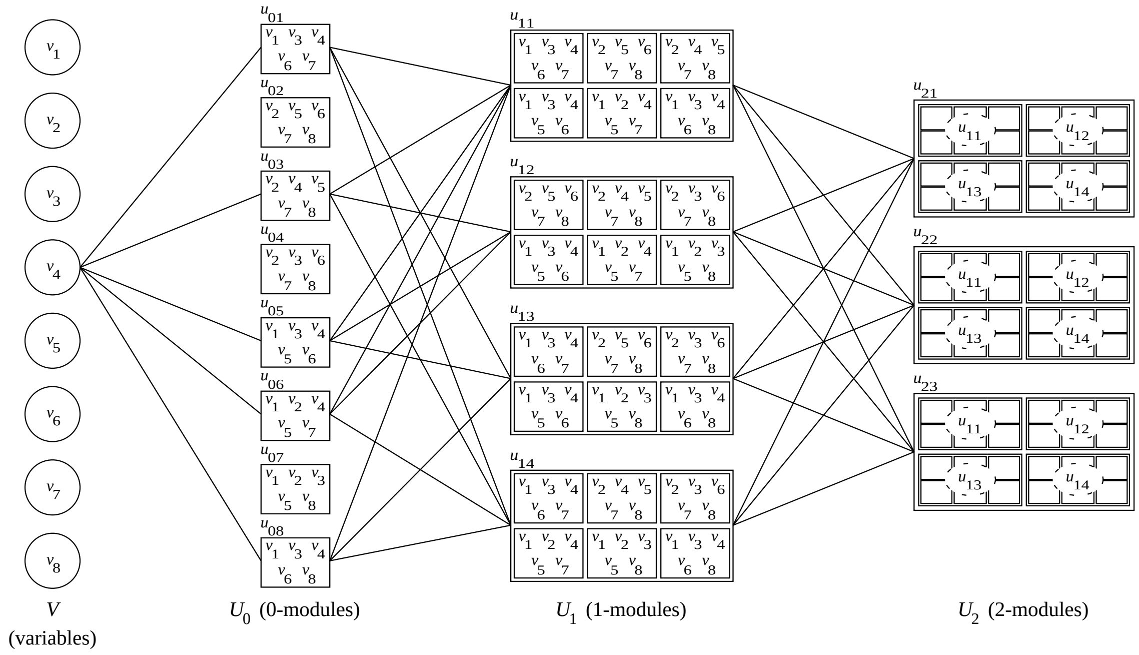

Figure 3.2: contents of i-modules, $0 \leq i \leq 2$, according to the HMOS of Figure 3.1. Variable v_4 is first replicated into $r = 5$ copies, which are assigned to 0-modules, then 3 copies of each such module are assigned to suitable 1-modules. Finally, each 1-module is replicated into 3 copies. As a whole, the process creates $5 \cdot 3^2 = 45$ copies of v_4.

module, viewed as an indivisible unit, are replicated into 3 copies, which are assigned to distinct i-modules, for $0 < i \leq k$. The above process will eventually create 3^{k-i} replicas of each i-module and $r3^k$ copies per variable. An example of the content of i-modules and of the whole memory is given by Figures 3.2 and 3.3. We will reserve the term *copy* to denote the replica of a variable, and i-*block* to denote the replica of an i-module. (Note that with this terminology k-modules and k-blocks coincide.) A k-module is composed of $(k-1)$-blocks, which in turn are composed of $(k-2)$-blocks, and so on. Finally, 0-blocks contain copies of variables.

The mapping between variables and 0-modules is represented by a bipartite graph (V, U_0), where each variable $v \in V$ is adjacent to the r 0-modules whose 0-blocks hold the copies of v. Similarly, the mapping between $(i-1)$-modules and i-modules is represented by a bipartite graph (U_{i-1}, U_i), $1 \leq i \leq k$, where each $(i-1)$-module

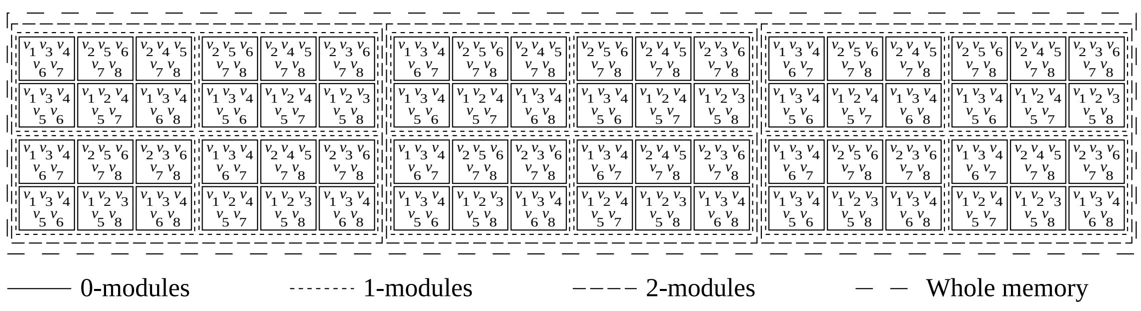

Figure 3.3: contents of the aggregate memory of the D-BSP according to the HMOS of Figure 3.1. The mapping process ensures that each i-block, $0 < i \leq k$, is recursively assigned to a distinct D-BSP cluster.

u is adjacent to the 3 i-modules whose i-blocks contain the $(i-1)$-blocks of u. We assume that $m = n^\tau$, for some constant $\tau > 1$. Let us fix $m_0 = |U_0| = n$ and $m_i = |U_i| = \Theta\left(n^{1/2^i}\right)$, for $1 \leq i \leq k$. We assume that the quantity $d_i = \lceil 3m_{i-1}/m_i \rceil$ is a power of 2, and that every $u \in U_i$ has degree at most d_i in (U_{i-1}, U_i), for $1 \leq i \leq k$. Later, we will show how the various graphs can be chosen to satisfy all of the above constraints on the parameters while exhibiting suitable expansion properties, which are needed to ensure low parallel access time.

The HMOS is mapped onto the D-BSP by assigning each i-block to a cluster of appropriate size in the following recursive fashion. Each of the m_k k-blocks is assigned to a distinct cluster with $t_k = n/2^{\lceil \log m_k \rceil}$ processors. The (at most) d_k $(k-1)$-blocks contained in a k-block, as prescribed by (U_{k-1}, U_k), are assigned to distinct subclusters of size $t_{k-1} = t_k/d_k$ of the cluster assigned to the k-block. In general, for $2 \leq i \leq k$, the (at most) d_i $(i-1)$-blocks contained in an i-block are assigned to distinct subclusters of size $t_{i-1} = t_i/d_i$ of the cluster assigned to the i-block. Consequently, an i-block, $1 \leq i \leq k$, is mapped to a cluster with

$$t_i = \frac{n}{2^{\lceil \log m_k \rceil} \prod_{j=i+1}^{k} d_j} \tag{3.1}$$

processors. Note that a certain number of processors may be *wasted* at each level of the mapping; a processor is wasted if no copies of variables are assigned to it. The presence of wasted processors is undesirable since it may lead to an asymptotic increase of storage space in the processors which are actually assigned some of the

$mr3^k$ copies of variables. Anyway, the following lemma shows this is not the case.

Lemma 3.3.1 *For $1 \le i \le k$, the mapping process assigns an i-block to a cluster containing*

$$t_i = \Theta\left(3^{i-k} n^{1-1/2^i}\right) \tag{3.2}$$

processors.

Proof If $i = k$ then the bound can be easily found by observing that $2^{\lceil \log m_k \rceil} = \Theta(m_k)$ and $m_k = \Theta\left(n^{1/2^k}\right)$. The case $1 \le i < k$ is a bit more involved since it is necessary to explicitly bound the quantity (3.1). The upper bound follows from the fact that $d_i \ge 3m_{i-1}/m_i$, therefore

$$t_i \le \frac{n}{m_k \prod_{j=i+1}^{k} 3m_{j-1}/m_j} = \frac{n}{m_k \cdot 3^{k-i} m_i/m_k} = O\left(\frac{n}{3^{k-i} m_i}\right).$$

The lower bound is obtained by leveraging on the inequality $d_i \le 3m_{i-1}/(m_i - 1)$, which, if n is sufficiently large, makes it possible to bound the product in (3.1) as:

$$\prod_{j=i+1}^{k} d_j = \Omega\left(3^{k-i}\frac{m_i}{m_k}\prod_{j=i+1}^{k-1}\frac{m_j}{m_j - 1}\right) = \Omega\left(3^{k-i}\frac{m_i}{m_k}\prod_{j=i+1}^{k-1}\left(1 + \frac{1}{n^{1/2^j} - 1}\right)\right). \tag{3.3}$$

Lengthy but straightforward calculations show that the product that appears in the right-hand side of (3.3) is upper bounded by a suitable constant, hence $t_i = \Omega\left(n/(3^{k-i}m_i)\right)$ and the thesis follows. □

This property shows that each of the $3^{k-i}m_i$ i-blocks is assigned to a cluster of asymptotically maximal size, so wasted processors and memory size increase are contained within constant factors. Moreover, t_i is a power of 2 and, since $k = O(\log\log n)$, we have that $t_i > 1$ if n is large enough.

The mapping of the HMOS is completed by evenly distributing the (at most) $d_1 = \Theta(\sqrt{n})$ 0-blocks contained in a 1-block among the $t_1 = \Theta\left(\sqrt{n}/3^k\right)$ processors of the cluster assigned to the 1-block; each such processor stores the copies of the variables contained in every 0-block it receives. In this way, a processor stores a total of $O\left(r3^k m/n\right)$ copies of variables, which ensures a balanced distribution of the copies among the processors.

As mentioned before, suitable *expansion properties* are required of the component graphs of the HMOS in order to ensure that, for any set of variables to be accessed, their copies be well spread among the processors' memories, thus yielding low access time.

Definition 3.3.2 *Let $G = (X, Y)$ be a bipartite graph where each node in X has degree d. For $0 < \sigma \leq 1$, $0 < \epsilon < 1$ and $1 \leq \mu \leq d$, G has (σ, ϵ, μ)-expansion if for any subset $S \subseteq X$, $S \leq \sigma|X|$, and for any set E of $\mu|S|$ edges, μ outgoing edges for each node in S, the set $\Gamma^E(S) \subseteq Y$ reached by the chosen edges has cardinality $|\Gamma^E(S)| = \Omega\left(|S|^{1-\epsilon}\right)$.*

We will choose (V, U_0) to have odd input degree r and output degree mr/n, and to exhibit $(n/m, \epsilon, (r+1)/2)$-expansion, where $\epsilon < 1$ is a suitable constant that will be chosen by the analysis. In [PPS00, Lemma 5.1], the existence of such a graph is proved for $\epsilon = 2(\tau - 1)/(r + 1)$ through the probabilistic method; however, explicit constructions are currently known for certain ranges of the parameters (see Section 3.5).

The main difference between the HMOS presented here and the one in [PPS00] concerns the structure of the graphs (U_{i-1}, U_i), $1 \leq i \leq k$. Specifically, we need these graphs to exhibit similar expansion and constructivity properties as in [PPS00], but the choice of the graphs is now influenced by further, stricter constraints. The main constraint is on the degree of the nodes in every bipartite graph (U_{i-1}, U_i) of the HMOS: to be precise the outdegree of each node $x \in U_{i-1}$ must be equal to the number of replicas of the $(i-1)$-module associated with x, i.e. it must be exactly 3; furthermore, the indegrees of the nodes in U_i must be powers of 2, and they must be all equal within constant factors. The last requirement is crucial for an efficient mapping of i-blocks to clusters; note that this requirement was not present in [PPS00] because the n-node mesh studied in that work exhibits a fine, regular interconnection topology that allows an efficient partitioning into submeshes even if such submeshes do not have exactly the same size and shape.

Regardless of the constraints imposed by D-BSP, it is still possible to obtain the bipartite graphs (U_{i-1}, U_i) by suitably manipulating BIBD graphs as in [PPS00]; however, the new scenario requires nontrivial modifications to the construction presented in that paper. A BIBD is formally defined as follows.

Definition 3.3.3 *A Balanced Incomplete Block Design [HJ86] with parameters z and q, or (z, q)-BIBD, is a bipartite graph (X, Y) such that $|Y| = z$, the degree of each node in X is q, and for any two nodes $y_1, y_2 \in Y$ there is exactly one node $x \in X$ adjacent to both.*

It is straightforward to see that a (z, q)-BIBD has $z(z-1)/(q(q-1))$ input nodes, and that the degree of each output node is $(z-1)(q-1)$. BIBDs are a good starting point for building the desired graphs, since they provide sufficient expansion. A proof of this fact is provided in [PPS00, Lemma 3.3 and Corollary 3.4]; for ease of presentation, we recall these results below.

Lemma 3.3.4 *Let (X, Y) be a (z, q)-BIBD. Consider a node $\bar{y} \in Y$ and a subset $S \subseteq X$ such that any node in S is adjacent to $\bar{y}$. Let E be a set of $\nu|S|$ edges, $\nu \leq q$, containing ν outgoing edges from each $x \in S$, and let $\Gamma^E(S)$ denote the set of nodes of Y reached by the chosen edges. Then, $|\Gamma^E(S)| \geq (\nu - 1)|S| + 1$.*

Corollary 3.3.5 *A (z, q)-BIBD has $(1, 1/2, \mu)$-expansion for every $2 \leq \mu \leq q$.*

By relying on the above results, the following lemma provides the crucial technical step through which the new graphs are obtained.

Lemma 3.3.6 *For every positive integer w, a bipartite graph $G = (X, Y)$ with $|X| = w$ can be explicitly constructed such that*

(i) $\sqrt{6w} < |Y| < 6 + 6\sqrt{6w}$*;*

(ii) every node of X has degree 3;

(iii) every node of Y has degree at most $d = \lceil 3w/|Y| \rceil$, and d is a power of 2;

(iv) for every subset $S \subset X$ whose nodes are all adjacent to some $y \in Y$ and for every selection of $\nu|S|$ edges, ν incident on every $s \in S$, the nodes of Y reached by the selected edges are at least $(\nu - 1)|S|/4 + 1$.

Proof The base G_{BIBD} of our construction is a $(z, 3)$-BIBD, where z is the smallest power of 3 such that $|A| \geq w$. By employing the techniques presented in [PPS00] we can extract a subgraph $G' = (X, B)$ of G_{BIBD} with $X \subseteq A$, $|X| = w$, and such that each node of X has degree 3 and each node of B has degree either $\lfloor 3w/z \rfloor$ or $\lceil 3w/z \rceil$.

Let d be the largest power of 2 not exceeding $\lfloor 3w/z \rfloor$, and let $B = \{b_i : 0 \leq i < z\}$. We index the edges of G' from 1 to $3w$ so that all edges incident on the same node b_i have consecutive indices, and, for $j < i$, edges incident on b_j have indices smaller than those incident on b_i. Observe that by the BIBD property, for every $0 < i < z$ there is at most one node $x \in X$ adjacent to both b_{i-1} and b_i: this allows us to rearrange the indices of the edges incident on b_i, for every i, so that if there are two edges (x, b_{i-1}) and (x, b_i), for some $x \in X$, the indices of these two edges differ by at least d. This is obtained by sequentially examining all pairs of nodes $\langle b_{i-1}, b_i \rangle$ starting from $\langle b_0, b_1 \rangle$: if there exists a pair of edges (x, b_{i-1}) and (x, b_i), for some $x \in X$, then the indexing is changed so that (x, b_{i-1}) is given the lowest index among the edges incident on b_{i-1}, and (x, b_i) is given the lowest index among the edges incident on b_i. Note that Step i of this procedure does not disrupt the work done in the previous steps: if for some $x' \in X$ there exists a pair of edges (x', b_{i-2}) and (x', b_{i-1}), which has been dealt with in Step $i - 1$, then the distance between the indices of these edges can only increase as a consequence of Step i.

We construct $G = (X, Y)$ by grouping the edges of G' into $\lceil 3w/d \rceil$ bundles of d consecutively indexed edges each (the last bundle may have less than d edges), and by creating a distinct node of Y for each bundle, which becomes the new right endpoint for all edges in the bundle.

Note that Properties (ii) and (iii) stated in the lemma follow easily from the above construction. To show that Property (i) holds, observe that the choice of z ensures that $w \leq |A| < z^2/6$, whence $|Y| \geq |B| = z > \sqrt{6w}$. Similarly, the upper bound on $|Y|$ follows from the inequalities $w > (z/3-1)^2/6$ and $|Y| \leq 2|B|$. Finally, the construction of the bundles guarantees that the edges incident on every node $y \in Y$ were previously adjacent to at most 2 nodes of B, therefore for every pair of nodes in Y there are no more than four nodes in X adjacent to both. By applying this observation to the nodes in S (that are all adjacent to a single node $y \in Y$) it is straightforward to show that for each $y' \in Y$, $y' \neq y$, there are at most 4 nodes of S adjacent to y'. Let now $\Gamma \subseteq Y$ be the set of nodes reached by the selected edges: from the previous observations, Γ has cardinality at least $(\nu - 1)|S|/4 + 1$. $\square$

As an immediate corollary of the above lemma, it is easy to prove that (U_{i-1}, U_i) still has good expansion properties.

Corollary 3.3.7 *The bipartite graph* (U_{i-1}, U_i) *has* $(1, 1/2, \mu)$*-expansion for* $1 \leq i \leq k$ *and* $\mu \in \{2, 3\}$.

Proof To prove that the corollary holds, it is enough to show that $|\Gamma^E(S)| > \sqrt{(\mu - 1)\mu|S|/4}$; by applying Property (iv) stated in Lemma 3.3.6, this inequality follows easily. $\square$

Given U_0, we apply Lemma 3.3.6 to construct the graph (U_0, U_1), thus fixing the size of U_1. Once the size of U_1 is known, we apply the lemma again to construct the graph (U_1, U_2), thus fixing the size of U_2. By iterating this process we can construct every graph (U_{i-1}, U_i), $1 \leq i \leq k$. By Property (i) of Lemma 3.3.6, such a process yields $m_i = |U_i| = \Theta\left(n^{1/2^i}\right)$, for $1 \leq i \leq k$, as required.

3.4 Access Protocol

Suppose that m shared variables are distributed among the n D-BSP processors according to the HMOS described above. In this section, we show how any n-tuple of variables can be efficiently accessed when every processor requires a read or write access to a distinct variable. Note that the case of concurrent accesses to the same variable can be reduced to the case of exclusive accesses by means of straightforward, sorting-based techniques which do not asymptotically affect the overall running time. The access protocol has the same structure as the one presented in [PPS00], but it differs in the implementation, which fully exploits the explicit hierarchical nature of the D-BSP model.

Let S denote the set of variables to be accessed, with every processor in charge of a distinct variable of S. As customary in redundant shared memory implementation schemes, a suitable set of copies for the variables in S must be chosen, so that accessing these copies will enforce data consistency and generate low memory contention and network congestion. We first explain how copy selection is accomplished and then show how the selected copies can be efficiently reached.

3.4.1 Copy Selection

The hierarchical structure of the HMOS provides a geographical distribution of the copies into the D-BSP clusters. Copy selection essentially aims at limiting the number of copies that have to be accessed in any block at any level of the HMOS, in order to reduce the traffic into/from the cluster storing the block. Consider a directed version $\mathcal{H}$ of the HMOS, where edges in every constituent bipartite graph are directed from the left to the right node set. Note that $\mathcal{H}$ is a dag and that the $r3^k$ copies of a variable v are in one-to-one correspondence with the source-sink paths of the subdag $\mathcal{H}_v \subset \mathcal{H}$ induced by v and by all of its descendants. In order to guarantee consistency, we require that the selected copies for every variable $v \in S$ form a *target set*, defined as follows.

Definition 3.4.1 *A set of copies C_v of v is a* target set *for v if, by coloring the nodes in $\mathcal{H}_v$ along the paths corresponding to the copies in C_v, it turns out that each colored node has a majority of its children colored.*

Specifically, $\mu = (r+1)/2$ children of the root v and two children of every colored node $x \neq v$ must be colored. It is easy to see that any two target sets for v share at least one copy. Consistency is enforced since a read access will always reach at least one most updated copy, which can be identified by means of timestamps [UW87].

Copy selection for the set of variables S is accomplished in $k+1$ iterations, numbered from 0 to k, during which in every $\mathcal{H}_v$, $v \in S$, the nodes along the paths corresponding to the copies being selected are colored level by level from the source to the sinks. Since, in general, a node of the HMOS belongs to several $\mathcal{H}_v$'s, we use a distinct color γ_v for each variable $v \in S$, and allow a node to be assigned a set of colors. Initially, the color set of every node is empty. Iteration 0 assigns the set $\{\gamma_v\}$ to the source v and adds γ_v to the color sets of μ of its adjacent 0-modules, for every $v \in S$. Iteration i, $0 < i \leq k$, selects two adjacent i-modules for every colored $(i-1)$-module u, and adds to their color sets the one assigned to u. (Clearly, if an i-module is selected for two or more colored $(i-1)$-modules, it receives the union of the color sets of these $(i-1)$-modules.) In this fashion, for every $v \in S$, at the end of Iteration i the nodes whose color sets contain γ_v form exactly $\mu 2^i$ distinct paths from the source to nodes at level i in $\mathcal{H}_v$. We call such paths *γ_v-colored paths.* We choose C_v as the set of copies of v corresponding to the $\mu 2^k$ γ_v-colored paths in $\mathcal{H}_v$ at the end of the last iteration. In order to identify the selected copies of each $v \in S$, the processor issuing the access request for v keeps track of the coloring being performed on $\mathcal{H}_v$.

We define the *weight* $w(u)$ of an i-module $u \in U_i$, $0 \leq i \leq k$, as the sum, over all $v \in S$, of the number of γ_v-colored paths containing u. The above coloring ensures that for every $u \in U_i$ with nonempty color set, only 2^{k-i} i-blocks of u contain copies in $\bigcup_{v \in S} C_v$, with exactly $w(u)$ copies per block. Using the expansion properties of the HMOS, we are able to establish suitably low bounds for the $w(u)$'s. The following lemmas provide such bounds and evaluate the running times of the iterations on D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$.

Lemma 3.4.2 *If (V, U_0) has $(n/m, \epsilon, \mu)$-expansion, Iteration 0 can be executed in time $O\left(n^\alpha + n^\beta\right)$, in such a way that, for every $u \in U_0$*

$$w(u) = O\left(n^\epsilon\right).$$

Proof At the beginning of the iteration, every processor P_v in charge of a variable $v \in S$ creates r packets of type $[P_v, v, u]$, where u is the name of a distinct 0-module adjacent to v in $\mathcal{H}_v$. Upon creation, all packets are regarded as *unmarked.* Then, the following steps are executed until μ packets for each variable are marked and all other packets are destroyed.

1. (*Sort*) Sort all unmarked packets by their third component (i.e., the associated 0-module).

2. (*Select*) Let a be a suitable constant. For each $u \in U_0$, if there are at most arn^ϵ packets with third component u in the sorted sequence, then all such

packets are marked. Otherwise, none of them is marked. Subsequently, all the packets are sent back to their origins.

3. (*Count*) For each $v \in S$, the total number of marked packets relative to v are counted. If these are at least μ, then exactly μ of them are kept and the remaining $\mu - 1$ (either marked or unmarked) are destroyed.

Finally, for every $v \in S$ the nodes in $\mathcal{H}_v$ corresponding to the μ marked packets picked for v receive color γ_v. Based on the expansion of (V, U_0) it can be shown [PPS00, Lemma 4.4] that at the end of the whole procedure the bound on $w(u)$ holds for every $u \in U_0$; moreover, after the i-th execution of the Sort, Select and Count steps, packets relative to at most $n/2^i$ variables remain unmarked. Hence, the i-th execution of these three steps can be implemented through r-sorting and prefix within an $(i-1)$-cluster, which yields the desired running time by applying the results of Propositions 2.4.1 and 2.4.2. □

Lemma 3.4.3 *For $1 \leq i \leq k$, Iteration i can be executed in time $O\left(2^i n^\alpha + n^\beta\right)$ in such a way that, for every $u \in U_i$*

$$w(u) = O\left(2^i n^{1-(1-\epsilon)/2^i}\right).$$

Proof We can adopt the same implementation of Iteration i as in [PPS00]. Let P_v be the processor in charge of variable v. At the beginning of Iteration i, P_v contains a certain number of packets of type

$$[P_v, v, u, h_{v,u}],$$

where u is a colored $(i-1)$-module and $h_{v,u}$ is the number of distinct colored paths from v to u; for each $v \in S$ the overall number of such paths is $\mu 2^{i-1}$. (If $i > 1$ the packets are received from the previous iteration, otherwise they are easily created by adding a fourth component $h_{v,u} = 1$ to the packets which survived Iteration 0.) Iteration i is composed of the following steps.

1. (*Sort*) Sort all packets by their third component (that is, by the associated $(i-1)$-module).

2. (*Create module-packets*) For each group of packets associated with an $(i-1)$-module $u \in U_{i-1}$, elect a *leader processor* P_u (for instance, choose as leader the processor with the lowest index among those containing packets in the group). Using a prefix algorithm, P_u calculates the weight

$$w(u) = \sum_{v \in S} h_{v,u},$$

then it creates 3 *module-packets*, which are auxiliary packets that will be destroyed at the end of the iteration. A module-packet is a 4-tuple

$$[P_u, u, \gamma(u,j), w(u)], \text{ for } j = 1, 2, 3,$$

where $\gamma(u,1)$, $\gamma(u,2)$ and $\gamma(u,3)$ identify the 3 i-modules adjacent to u.

3. (*Sort module-packets*) Sort the module-packets lexicographically by their third and fourth components.

4. (*Select module-packets*) For each i-module $x \in U_i$, select a maximal set M_x of module packets with third component x so that

$$\sum_{[P_u,u,x,w(u)] \in M_x} w(u) \leq c\mu 2^{i-1} \left(n^{1-(1-\epsilon)/2^i} + n^{1-(1-\epsilon)/2^{i-1}} \right),$$

with c a suitable constant that will be fixed later. Note that M_x has at most 3 elements. When the sets have been prepared, all the module-packets are sent back to the processors which created them, then each leader processor P_u counts the number h_u of its packets that have been selected: if $h_u < 2$, then P_u selects $2-h$ extra module-packets from the $3-h$ that were not chosen before, otherwise, P_u picks out 2 module-packets among those previously selected. In both cases, at the end of the procedure each leader processor owns exactly 2 marked packets: we can denote them with $[P_u, u, \gamma(u,j_1), w(u)]$ and $[P_u, u, \gamma(u,j_2), w(u)]$.

5. (*Append*) Send the labels $\gamma(u,j_1)$ and $\gamma(u,j_2)$ to all processors in the group of P_u, then destroy the module-packets and append such labels to the original packets. Finally, send all packets back to the processors that contained them at the beginning of the first (*Sort*) step.

6. (*Color*) For each modified packet $[P_v, v, u, h_{v,u}, \gamma(u, j_1), \gamma(u, j_2)]$ received by processor P_v, add color γ_v to the color sets of the i-modules $\gamma(u, j_1)$ and $\gamma(u, j_2)$. Moreover, for each newly marked node $u' \in U_i$ create a new packet $[P_v, v, u', h_{v,u'}]$, where $h_{v,u'}$ is the sum of the $h_{v,u}$'s in all the modified packets which contain u' as the fifth or sixth component. The newly-created packets are passed to the next iteration, while the old ones are discarded.

Iteration i essentially requires a constant number of $O(2^i)$-sorting, broadcast and prefix operations; the running time follows from Propositions 2.4.1 and 2.4.2. The bound on $w(u)$ is obtained by repeating the argument in [PPS00, Lemma 4.5] and tuning the constants to reflect the different expansion properties of the graphs defined in Lemma 3.3.6 with respect to the ones adopted in that paper; to be precise, it can now be proved that there is a suitable constant $c \geq 12$ such that, at the end of Iteration i, $w(u) \leq c\mu 2^i n^{1-(1-\epsilon)/2^i}$ for each $u \in U_i$. □

The following theorem is an immediate consequence of the above lemmas and the preceding discussion.

Theorem 3.4.4 *Copy selection requires time $O\left(2^k n^\alpha + kn^\beta\right)$ and ensures that every i-block, $0 \leq i \leq k$, contains at most $O\left(2^i n^{1-(1-\epsilon)/2^i}\right)$ selected copies.*

3.4.2 Access to the Selected Copies

After copy selection is completed, for every $v \in S$ the processor in charge of v creates $\mu 2^k$ distinct messages to access the copies in C_v. Each message is routed to its destination (i.e., the processor whose memory stores the requested copy) where the read/write access is performed. (For the case of read accesses, the return of the accessed data to the requesting processors is dealt with in a symmetric fashion, hence we omit its description.) Messages are delivered to the destinations in $k+1$ stages through smaller and smaller clusters, thus taking advantage of the good distribution of the copies among the i-blocks, for every i.

For $1 \leq i \leq k$, let $\tau_i = \log n - \log t_i$ and recall that every i-block is assigned to a distinct cluster with t_i processors, that is, a τ_i-cluster. Let also $\tau_{k+1} = 0$. The stages are numbered from $k+1$ down to 1. For $k+1 \geq i \geq 2$, Stage i is executed in parallel and independently in every τ_i-cluster and sends all messages residing in the cluster to arbitrary positions in the internal τ_{i-1}-clusters associated with their destination $(i-1)$-blocks, in such a way that the processors in the same τ_{i-1}-cluster receive approximately the same number of messages. The intermediate destination of each message is easily established via sorting and ranking. Finally, in Stage 1 every message is sent to its final destination, in parallel and independently within every τ_1-cluster.

Let δ_i be the maximum number of messages held by any processor at the beginning of Stage i, $k+1 \geq i \geq 1$. By virtue of the message balancing described above, δ_i is upper bounded by the maximum number of selected copies within a single i-block divided by the size of a τ_i-cluster. Moreover, at the end of Stage 1, $O(n^\epsilon)$ messages per 0-block reach the processor storing the block in its local memory. Since the number of 0-blocks assigned to a single processor is $\Theta\left(3^k\right)$, the maximum number of messages delivered to a processor at the end of the access protocol is $\delta_0 = O\left(3^k n^\epsilon\right)$. By plugging in the bounds derived in Theorem 3.4.4 and Equation (3.2), we obtain:

$$\delta_i = \begin{cases} \mu 2^k & \text{for } i = k+1, \\ O\left(2^i 3^{k-i} n^{\epsilon/2^i}\right) & \text{for } k \geq i \geq 0. \end{cases}$$

Stage i can be easily implemented by means of a constant number of δ_i-sorting and prefix operations and one instance of (δ_i, δ_{i-1})-routing within τ_i-clusters. From Propositions 2.4.1, 2.4.2 and 2.4.3 it then follows that the time required to access the selected copies is

$$O\left(\left(2^k n^{\frac{(1-\alpha)\epsilon}{2^k}} + \sum_{i=1}^{k} 2^i 3^{(1-\alpha)(k-i)} n^{\frac{(2-\alpha)\epsilon-\alpha}{2^i}}\right) n^\alpha + kn^\beta\right). \tag{3.4}$$

Note that the above time always dominates over the time for copy selection. Let us now choose r so that the expansion parameter ϵ of (V, U_0) is strictly less than $\alpha/(2-\alpha)$. Simple manipulations of Equation (3.4) then suffice to prove that an n-tuple of variables can be accessed by the D-BSP processors in time

$$O\left(2^k n^{\alpha + \frac{\alpha(1-\alpha)}{2^k(2-\alpha)}} + kn^\beta\right). \tag{3.5}$$

In order to achieve the best possible tradeoff between performance and redundancy on a specific D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$, we need to choose a suitable value for k as a function of α and β. More specifically, when $\alpha < \beta$, it suffices to choose $k = O(1)$ large enough to obtain optimal $O\left(n^{\beta}\right)$ slowdown with constant redundancy. This result demonstrates that optimal worst-case slowdown is achievable on machines where delays due to latency dominate over those due to bandwidth. Instead, when $\alpha \geq \beta$, by choosing $k = \log\log n$, we obtain a minimal slowdown of $O(n^{\alpha}\log n)$ with redundancy $\Theta\left(\log^{\log 3} n\right) = O\left(\log^{1.59} n\right)$.

3.5 Constructivity Issues

It must be remarked that the performance of the access protocol analyzed in the previous section relies on the $(n/m, \epsilon, \mu)$-expansion of (V, U_0). Although such level of expansion is considerably milder than those required by most schemes in the literature (e.g., $\epsilon = O(1/\log n)$ in [UW87, AHMP87, HPP01]), in general we can only show the existence of (V, U_0) through the probabilistic method. However, unlike the aforementioned schemes, ours can take advantage of the few explicit constructions known in the literature. For example, in [PP97] an efficient construction is provided for a bipartite graph with m inputs, $n' = \Theta\left(m^{2/3}\right)$ outputs, input degree $r = 3$, and output degree $\Theta\left(m^{1/3}\right)$, which exhibits $(1, 1/3, 2)$-expansion. This graph can be employed to implement (V, U_0) for every value $m = O\left(n^{3/2}\right)$. Note that for $m = o\left(n^{3/2}\right)$, the number n' of output nodes in the graph is $o(n)$. In this case, in order to obtain $|U_0| = n$, we simply replace each output node by n/n' new outputs, subdividing the incoming edges evenly among these new nodes. Clearly, the expansion property is not affected by this modification. Also, it is easy to see that by plugging $\epsilon = 1/3$ and $\alpha \geq 1/2$ in Equation (3.4), the overall running time of the access protocol reduces to the one stated in Equation (3.5), thus yielding the constructivity result of Theorem 3.2.1.

3.6 An Optimal Randomized Scheme

As we have already said, randomized schemes (see e.g., [CMadHS00, Ran91]) usually distribute the variables randomly among the memory modules local to the processors. As a consequence of such a scattering, a simple routing strategy is sufficient to access any n-tuple of variables efficiently, with high probability. Following this line, we can give a simple, randomized scheme for shared memory access on D-BSP. Assume, for simplicity, that the variables be spread among the local memory modules by means of a totally random function. As a matter of fact, a polynomial hash function drawn from a $\log n$-universal class [CW79] suffices to achieve the same results [MV84], but it takes $\text{poly}(\log n)$ rather than $O(n \log n)$ random bits to be generated and stored at the nodes. We have:

Theorem 3.6.1 *Any n-tuple of memory accesses on D-BSP$(n, \boldsymbol{g}, \boldsymbol{\ell})$ can be performed in time*

$$O\left(\sum_{i=0}^{\lfloor \log(n/\log n)\rfloor - 1} T_{\text{PREFIX}}(i) + g_{\lfloor \log(n/\log n)\rfloor} \frac{\log n}{\log\log n} + \ell_{\lfloor \log(n/\log n)\rfloor}\right) \tag{3.6}$$

with high probability, where $T_{\text{PREFIX}}(i)$ denotes the time of a prefix-sum operation within an i-cluster.

Proof Consider the case of write requests. By regarding the D-BSP clusters as bins, a standard occupancy argument [MR95] suffices to show that, during an access to n randomly distributed variables, there will be no more than $O(|C|)$ messages destined to the same cluster C for $|C| \geq \log n$, and no more than $O(\log n/\log\log n)$ messages destined to a single processor, with high probability. According to these observations, the access is performed in $\lfloor \log(n/\log n)\rfloor + 1$ steps by first sending messages to their destination cluster of size $\log n$ in a balanced way, then sending each message to the appropriate processor in a final routing step. More specifically, in Step i, for $1 \leq i \leq \lfloor \log(n/\log n)\rfloor$, the messages containing the access requests are sent to their destination i-clusters, so that they are evenly distributed among the processors of the cluster: each step requires a constant number of prefix operations and the routing of an $O(1)$-relation in $(i-1)$-clusters. In the last step, the messages are directly sent to their final destinations, where the memory access is performed: by the occupancy argument, the degree of the relation in this case is $O(\log n/\log\log n)$, with high probability. The running time stated in Equation 3.6 is obtained by simply summing up the costs of the operations performed in the various steps of the algorithm.

For read accesses, the return journey of the messages containing the accessed values can be performed by reversing the algorithm for writes, thus remaining within the same time bound. □

By plugging in Equation (3.6) the time for prefix given by Proposition 2.4.1, we obtain:

Corollary 3.6.2 *Any n-tuple of memory accesses can be performed in optimal time $O\left(n^\alpha + n^\beta\right)$ on D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$.*

Observe that under a uniform random distribution of the variables among the memory modules, $\Theta\left(\log n / \log\log n\right)$ out of *any* set of n variables will be stored in the same memory module, with high probability, hence any randomized access strategy will require at least $\Omega\left(n^\alpha \log n / \log\log n + n^\beta\right)$ time on a BSP with bandwidth n^α and latency n^β. This lower bound proves that the network locality featured by D-BSP$(n, \boldsymbol{g}^{(\alpha)}, \boldsymbol{\ell}^{(\beta)})$ affords a better simulation slowdown.

Chapter 4

Parallelism and Temporal Locality

Most modern, high-performance computer platforms consist of several processors, each endowed with its local memory, communicating through some interconnection. When looking at the overall aggregate memory available at the system, one observes a multi-level hierarchical structure, where lower (fast) levels are internal to the processing nodes while upper (slow) levels are implicitly imposed by the interconnection topology. This is the case, for example, of popular architectures such as clusters of workstations or UMA and NUMA multiprocessors. The main idea behind the efficient exploitation of such a hierarchical organization is that an algorithm will be able to reduce access costs

- by organizing the computation so that the same data be frequently reused within a short time interval (*temporal locality* of reference), and that consecutive data in memory be involved in consecutive operations (*spatial locality*);

- by organizing communication so that each processor exchange data mainly with its near neighbors (*network locality*).

The performance of applications running on these platforms is determined by the maximum number of operations performed at a node (*computation*), and by the data movements required to bring the data close to the units that have to process them. These movements can either occur at individual nodes (*local accesses*) or involve several nodes (*communication*).

In the past, the development of fast algorithms has either focused on minimizing memory access time on sequential hierarchies, or on attaining optimal computation/communication tradeoffs on parallel architectures with flat local memories. Moreover, in the latter case, high parallelism has been often pursued under the assumption that performance could then be scaled down linearly with the number of processors. Such is the case, for instance, of PRAM algorithms designed for a large number of virtual processors (typically proportional to the input size), or algorithms for bulk-synchronous models such as BSP [Val90a] and CGM [DFRC96], a BSP variant where the nodes have a fixed amount μ of memory and every communication is regarded as a $\Theta(\mu)$-relation. In these studies, virtual parallelism is employed to hide communication latency. It is well known, however, that when implementing an algorithm designed for many (virtual) processors on a real machine with a fixed number of processors, in addition to the natural slowdown due to the loss of parallelism (Brent's lemma [Bre74]), a further slowdown may be introduced if the (larger) subcomputations entrusted to each individual processor exhibit a low degree of locality of reference. This chapter offers a rigorous framework integrating parallelism and memory hierarchy, and showing that in many cases this additional slowdown can be avoided by designing algorithms that expose parallelism in the structured fashion prescribed by D-BSP. The focus is on temporal locality; a complete scenario including spatial locality will be developed in Chapter 5.

The rest of the chapter is organized as follows. Section 4.1 summarizes existing results concerning parallelism and locality of reference. Section 4.2 gives an overview of our original contributions on the subject. Section 4.3 defines our reference models. Section 4.4 describes the simulation scheme, first introducing its main ideas with two special cases (Subsections 4.4.1 and 4.4.2), and then deriving the general result (Subsection 4.4.3). Finally, in Section 4.5 we provide evidence of how our simulation yields efficient hierarchy-conscious sequential algorithms from efficient parallel ones.

4.1 Previous Work

As mentioned before, few works in the literature have dealt with both parallelism and memory hierarchy in an integrated fashion. The *Parallel Hierarchical Memory Model*

(P-HMM), defined in [VS94b], is a generalization of the sequential HMM model by [AACS87] consisting of p HMM processors communicating through some powerful interconnection attached to the fastest levels of the local hierarchies. A similar model, also featuring block transfer, is the *Parallel Memory Hierarchy* (PMH) model of [ACF93]. Unlike P-HMM, in PMH communication occurs via a shared memory module which is seen as connected to one of the slowest levels of each hierarchy. Both the P-HMM and the PMH models idealize communication either by assuming that its cost never dominates the time complexity (P-HMM) or by disregarding the impact of the interconnection topology (PMH). Another attempt at integrating parallelism and memory hierarchy is the H-PRAM [HR92], where processors can be partitioned into clusters operating as independent PRAM machines with smaller memories. However, in the H-PRAM access costs do not depend on the position of the data in memory, but solely on the size of the cluster, hence locality of reference is not captured.

The $M_d(n, p, m)$ model introduced by Bilardi and Preparata [BP97, BP99] features a better integration of parallelism and memory hierarchy and explicitly describes the network's topology. Specifically, an $M_d(n, p, m)$ is a d-dimensional mesh of p HMM nodes where the memory at each node has size nm/p, and access function $f(x) = \lceil (x + 1)/m \rceil^{1/d}$. The cost for sending a constant-size message from a node to a neighbor is proportional to the cost of accessing the farthest cell in the node's local memory. Although the authors argue that the $M_d(n, p, m)$ is the only scalable architecture under speed of light limitations, its reliance on a specific interconnection may hamper the portability and generality of algorithm design based on this model.

In [BP97] it is shown that an $M_d(n, n, m)$ can be simulated by an $M_d(n, p, m)$, for $p < n$ and $d = 1, 2$, with slowdown $(n/p)\Lambda(n, p, m)$. This provides an analogue of Brent's lemma except for the factor $\Lambda(n, p, m)$, which represents an extra slowdown due to the interaction with the larger local memories. Such a slowdown, which can grow up to $(n/p)^{1/d}$, is proved to be unavoidable for certain computations [BP99]. Note that the $M_d(n, n, m)$ being simulated is *fine-grained* in the sense that each node has only a constant number of available memory cells (i.e., access time can be regarded as a constant).

The translation of parallelism into locality of reference is also studied in [DDH03, DDHM99, SK97]. These works present strategies for efficiently simulating BSP-like computations designed for machine configurations with flat local memories on smaller configurations where a two-level (disk-RAM) hierarchy is provided at each node. The key of the simulation is a careful exploitation of the disks through striped access to the data.

4.2 Original Contributions

Continuing along the lines of [BP97, BP99, DDH03, DDHM99], in this chapter we show that for a wide class of computations, parallelism exposed in a structured fashion can be fully translated into temporal locality, affording the formulation of a Brent-like result proving that for such computations no slowdown due to the loss of locality is incurred when scaling down the number of processors.

Again, our results lay their foundations on the D-BSP model. To deal with temporal locality, we modify the D-BSP definition we gave in Section 2.2 by regarding each processor as an HMM and setting the cost of communication within a cluster to be proportional to the cost of accessing the farthest cell in a memory of size equal to the aggregate size of the cluster processors' memories. In this fashion, we integrate memory hierarchy and network by regarding the latter as a seamless continuation of the former.

Our main technical result is a uniform scheme that simulates any v-processor D-BSP computation on a v'-processor D-BSP, with $v' \leq v$ and the same aggregate memory size. For a large family of computations, including most prominent ones (e.g., sorting, FFT and matrix multiplication), our simulation exhibits an optimal $O(v/v')$ slowdown, thus providing an analogue of Brent's lemma. The simulation is based on a recursive strategy aimed at translating D-BSP submachine locality into temporal locality on the HMM. The strategy is similar in spirit to the one employed in [BP01] for porting DAG computations efficiently across sequential memory hierarchies while retaining temporal locality.

Our results complement those in [BP97, BP99] since they show that the superlinear slowdown incurred by the simulation of fine-grained algorithms can only manifest itself for a restricted class of machine-dependent computations, namely those which take the fullest advantage of fine topological details. For these computations, some degree of network locality is bound to be lost when transformed into temporal locality.

Finally, as an important special case, by setting $v' = 1$ our simulation can be employed to obtain efficient hierarchy-conscious sequential algorithms from efficient fine-grained ones. In this fashion, a large body of algorithmic techniques exhibiting structured parallelism can be effortlessly transferred to the realm of sequential algorithms for memory hierarchies. We provide evidence of this fact by showing that, for a number of prominent computations, optimal sequential algorithms can be obtained in this fashion. In this respect, our work provides a generalization of the results by [DDH03, DDHM99] to multi-level memory hierarchies.

Recall that in memory hierarchies another type of locality exists, namely *spatial locality*, exploitable through the block transfer mechanism. The issue of how to include spatial locality into our framework will be addressed in Chapter 5.

4.3 Machine Models

4.3.1 HMM

The *Hierarchical Memory Model* (HMM) was introduced in [AACS87] as a random access machine where access to memory location $x \geq 0$ requires time $f(x)$, for a given non-decreasing function $f(\cdot)$[1]. It is supposed that an n-ary operation (i.e., an operation involving memory cells $x_1, \ldots, x_n$) can be completed in time $1 + \sum_{i=1}^{n} f(x_i)$, regardless of the value of n. We refer to such a model as $f(x)$-HMM. As most works in the literature, we focus our attention on *polynomially bounded* access functions.

Definition 4.3.1 *A non-decreasing function $f(x)$ is said to be polynomially bounded if there exists a constant c such that $f(2x) \leq cf(x)$, for any x.*

[1]If necessary, we will silently assume that $f(x)$ is rounded to the nearest integer.

Particularly interesting and widely studied special cases are the polynomial function $f(x) = x^\alpha$ and the logarithmic function $f(x) = \log x$. The following two technical facts are proved in [AACS87].

Fact 4.3.2 *If $f(x)$ is polynomially bounded, then the amount of memory available for an $f(x)$-HMM algorithm can be doubled while increasing the running time at most by a constant factor.*

Fact 4.3.3 *If $f(x)$ is polynomially bounded, then the time to access the first n memory cells of an $f(x)$-HMM is $\Theta(nf(n))$.*

4.3.2 D-BSP with Hierarchical Memory

To deal with the memory hierarchy in a reasonable way, we equip the original D-BSP model with hierarchical memory at the processors; a single cost function accounts for both memory accesses and communication costs in a uniform fashion. We will refer to this model as a D-BSP$(v, \mu, f(x))$.

Let v be a power of two. A D-BSP$(v, \mu, f(x))$ is a collection of v processors $\{P_j : 0 \leq j < v\}$ communicating through a router, where each processor is an $f(x)$-HMM machine with memory size μ. In particular, a D-BSP$(1, \mu, f(x))$ coincides with an $f(x)$-HMM with memory size μ. As in the original D-BSP, the v processors are partitioned into 2^i fixed, disjoint i-clusters, $0 \leq i \leq \log v$, of $v/2^i$ processors each; the processors of a cluster are capable of communicating among themselves independently of the other clusters. The clusters form a hierarchical, binary decomposition tree of the D-BSP machine. A D-BSP$(v, \mu, f(x))$ program consists of a sequence of labelled supersteps; the type and order of operations during an i-superstep, $0 \leq i \leq \log v$, are the same which we have already described in Section 2.2 for the original D-BSP. If each processor spends at most τ units of time performing local computation during the superstep, and if the messages that are sent form an h-relation (i.e., each processor is the source or destination of at most $h > 0$ messages), then the cost of the i-superstep is upper bounded by

$$\tau + f(\mu v/2^i) \cdot h.$$

We call a program *full* if the communication required by every superstep is a $\Theta(\mu)$-relation, that is, each processor sends/receives an amount of data proportional to its local memory size. As we will see, several prominent problems can be efficiently solved by full programs. Also, it is reasonable to assume that any D-BSP computation ends with a global synchronization (i.e., the last superstep executed by a D-BSP processor is always a 0-superstep).

Note that the communication costs in our variant of the model are as in a standard D-BSP where both bandwidth and latency parameters within i-clusters are set to $f(\mu v/2^i)$. Our choice for such parameters aims at creating a seamless hierarchy of memory access and communication costs. More specifically, the communication medium is regarded as an extension of the local memory hierarchies, with each message sent by a processor in an i-cluster C being charged with the cost of accessing the farthest memory cell in an $f(x)$-HMM with memory size equal to the aggregate memory size of C.

Although in this chapter we deal with arbitrary polynomially bounded access functions $f(x)$, we will support our findings by considering, as case studies, the aforementioned polynomial and logarithmic functions. For the polynomial function $f(x) = x^\alpha$, we observe that when $\alpha = 1/d$, the D-BSP(v, μ, x^α) can be regarded as an abstraction of the d-dimensional architecture proposed by Bilardi and Preparata in [BP97] as the only scalable machine in the limiting technology. In fact, this D-BSP can be simulated on such an architecture with constant slowdown.

4.4 The General Simulation Algorithm

In this section, we present a general scheme to simulate any D-BSP$(v, \mu, f(x))$ program on a D-BSP$(v', \mu v/v', f(x))$, with $v' \leq v$. In order to introduce the main ideas underlying our simulation strategy, we first consider the simpler case $v' = 1$ and $f(x) = x^\alpha$, with $\alpha < 1$ a positive constant (Subsection 4.4.1). We then describe suitable modifications that are needed to extend the simulation to the case of arbitrary polynomially bounded functions (Subsection 4.4.2). Finally, in Subsection 4.4.3 we remove the assumption $v' = 1$.

4.4.1 Simulation of D-BSP(v, μ, x^α) on x^α-HMM

Without loss of generality, we can restrict ourselves to consider D-BSP(v, μ, x^α) programs where the labels of consecutive supersteps differ by at most 1: we refer to these programs as *smooth*. In a smooth program, an i-superstep can only be followed by a j-superstep, with $|j - i| \leq 1$. Indeed, for any non-smooth program there is an equivalent smooth program which exhibits the same asymptotic time complexity. This is easily seen as follows. If $f(x) = x^\alpha$, the cost of communication grows polynomially with the cluster size, hence one can insert, between any two consecutive supersteps with labels i and j, with $|j - i| > 1$, one dummy superstep for every distinct label between i and j. This transformation increases the time complexity of the program by at most a constant multiplicative factor.

Consider now a smooth D-BSP(v, μ, x^α) program $\mathcal{P}$ to be simulated on an x^α-HMM. Let us regard the HMM memory as divided into v *blocks*, numbered from 0 to $v-1$, of μ cells each, with block 0 at the top of memory (i.e., comprising memory cells $0, 1, \ldots, \mu - 1$), and block $v - 1$ at the bottom. During the course of the simulation every block will contain the *context* of a distinct D-BSP processor, that is, an image of the processor's local memory at a certain point of execution. At the beginning of the simulation, block i contains the context of processor P_i, $i = 0, 1, \ldots, v - 1$, but this association changes as the simulation proceeds. Additional constant-size space is needed for bookkeeping operations. For simplicity, we assume that $O(1)$ registers with unit-time access are employed for this purpose. Alternatively, we could use the memory cells at the top of memory and shift the processors' contexts down by a constant amount, thus paying a negligible time penalty.

The simulation of $\mathcal{P}$ on HMM is organized in a number of *rounds*, where a round simulates the operations prescribed by a certain superstep of $\mathcal{P}$ for a certain cluster, and performs a number of context swaps to prepare the execution of the next round. Suppose that the supersteps of $\mathcal{P}$ are numbered consecutively and let i_s be the label of Superstep s, for $s \geq 0$ (i.e., Superstep s is executed independently within each i_s-cluster). The simulation algorithm is given by the following pseudo-code, where the loop iterations correspond to rounds.

```
        **while** true **do**
1           $P \leftarrow$ processor whose context is on top of memory
            $s \leftarrow$ superstep number to be simulated next for $P$
            $i_s \leftarrow$ superstep index to be simulated next for $P$
            $C \leftarrow i_s$-cluster containing $P$
2           Simulate Superstep $s$ for cluster $C$
3           **if** $P$ has finished its program **then exit**
            {final 0-superstep executed: simulation complete}
4           **if** $i_{s+1} = i_s - 1$ **then**
                Swap the first $|C|$ blocks on top of memory
                with the next $|C|$ blocks
```

During the course of the simulation we say that a D-BSP processor P is *s-ready* if for all processors in the i_s-cluster of P (including P itself) Supersteps 0, 1, . . . , $s-1$ have been simulated while Superstep s has not yet been simulated. As will be proved later, the following two invariants are maintained at the beginning of each round. Let s and C be defined as in Step 1 of the round.

INV1 The contexts of all processors in C are stored in the topmost $|C|$ blocks, sorted in increasing order by processor number. All processors in C are s-ready.

INV2 For any other cluster C', the contexts of all processors in C' are stored in consecutive memory blocks.

If invariant **INV1** holds, the cluster simulation in Step 2 can be correctly performed as follows.

```
        **for** $j \leftarrow 0$ **to** $|C| - 1$ **do**
2.1         Swap the contents of memory blocks 0 and $j$
2.2         Simulate the local operations prescribed by Superstep $s$
            for the processor whose context is in block 0
2.3         Swap the contents of memory blocks $j$ and 0
2.4     Simulate the message exchange prescribed by
        the superstep for cluster $C$
```

The message exchange in Step 2.4 can be completed by scanning the output pools of the processor contexts sequentially and delivering each message to the input pool in the destination processor's context. Since by invariant **INV1** the contexts of the processors are sorted by processor number, the location of each input pool is easily determined based on the processor number.

Theorem 4.4.1 *The simulation algorithm is correct.*

Proof We first show that the invariants **INV1** and **INV2** hold at the beginning of each round: this is proved by induction on the number v of D-BSP processors. The claim is trivial for the basis $v = 1$, since in this case, the D-BSP program is simply a sequence of 0-supersteps, which are simulated in a straightforward fashion one after the other. Suppose that the claim is true for machines of up to v processors, and consider the simulation of a program $\mathcal{P}$ of t supersteps for a D-BSP with $2v$ processors. First, consider the case in which the t supersteps include a single 0-superstep (which by our former assumption must be the last superstep). If $t = 1$ then the claim trivially follows. Otherwise, consider Step 4 of the algorithm's pseudo-code: the algorithm swaps the contexts of cluster $C_0^{(1)}$ with those of the sibling cluster $C_1^{(1)}$, thus touching memory blocks $v/2, \ldots, v-1$, only if $i_{s+1} = 0$. Since $i_s > 0$ for $s < t$, such a swap never takes place before Superstep t, and the algorithm initially simulates the first $t-1$ supersteps for $C_0^{(1)}$ and its subclusters as if it were running a program for a D-BSP with v processors. By the inductive hypothesis, the two invariants hold for all the rounds performed in this initial phase, which ends with a round that simulates superstep $t-1$ (a 1-superstep, by the smoothness of $\mathcal{P}$) for cluster $C_0^{(1)}$. At the end of such a round, the simulation algorithm swaps the contexts of the processors in $C_0^{(1)}$ with those in the sibling cluster $C_1^{(1)}$. Since this is the first time $C_1^{(1)}$ is brought to the top of the HMM memory and the first superstep of $\mathcal{P}$ has label strictly greater than 0, both invariants hold at the beginning of the next round. The inductive hypothesis can then be applied again for $C_1^{(1)}$, thus showing that the invariants hold for all rounds up to and including the round that simulates superstep $t-1$ for $C_1^{(1)}$. At the end of this latter round, $C_0^{(1)}$ and $C_1^{(1)}$ are swapped back and the next (final) round simulates Superstep t, which, being a 0-superstep, involves the entire machine. Clearly, the invariants hold at the beginning of the final round. If $\mathcal{P}$ contains more than one 0-superstep, we can split the program into subprograms terminating at 0-superstep boundaries and iterate the above argument for each subprogram.

Once **INV1** and **INV2** are proved to hold, it is easy to see that the simulation algorithm executes, at the rate of one cluster per round, all the clusters (and, consequently, all the supersteps) that constitute program $\mathcal{P}$. The algorithm terminates when the topmost processor P has finished its program: since the last superstep of P is necessarily a 0-superstep, all the other processors of the guest D-BSP machine have finished their program as well. The correctness of the entire algorithm immediately follows. $\square$

Let us now evaluate the running time of a generic round of the algorithm, where a given superstep s for a given i_s-cluster C is simulated. Clearly, Steps 1 and 3 require constant time. Consider now Step 2, and note that, for Substep 2.2, the simulation of the local operations of each processor $P_j \in C$ incurs no slowdown: in fact, each D-BSP processor is an x^α-HMM, and the context of P_j is on top of the HMM memory when local computation takes place. Note also that by virtue of invariant **INV1**, the time for the message exchange in Substep 2.4 is bounded by the cost of accessing the first $\mu|C|$ HMM memory cells a constant number of times. As a whole, the same bound holds for Substeps 2.1 and 2.3. Hence, letting τ_s denote the maximum local computation time for a processor in Superstep s, the simulation of Step 2 is accomplished in time

$$O\left(\tau_s|C| + \sum_{x=0}^{\mu|C|-1} x^\alpha\right) = O\left(|C|\left(\tau_s + \mu(\mu|C|)^\alpha\right)\right)$$

because of Fact 4.3.3. Finally, observe that whenever a swap between $|C|$ blocks occurs in Step 4, we have just finished simulating the computation of a cluster of $|C|$ processors, hence the cost $\Theta\left((\mu|C|)^{\alpha+1}\right)$ required by such swap on an x^α-HMM is dominated by that of Step 2.

By summing up the results of the analysis, the overall running time for the simulation of a given i_s-cluster is

$$O\left(\frac{v}{2^{i_s}}\left(\tau_s + \mu(\mu v/2^{i_s})^\alpha\right)\right).$$

Since Superstep i_s is composed by 2^{i_s} such clusters, the overall time for an entire i_s-superstep is

$$O\left(v\left(\tau' + \mu(\mu v/2^{i_s})^\alpha\right)\right),$$

where τ' is the maximum local computation time for a processor during the superstep. By considering the contributions of all the supersteps in program $\mathcal{P}$, the equation above immediately leads to the following theorem.

Theorem 4.4.2 *Consider a program $\mathcal{P}$ for D-BSP(v, μ, x^α), where each processor performs local computation for $O(\tau)$ time, and there are λ_i i-supersteps, for $0 \leq i \leq \log v$. Then, $\mathcal{P}$ can be simulated on x^α-HMM in time*

$$O\left(v\left(\tau + \mu \sum_{i=0}^{\log v} \lambda_i(\mu v/2^i)^\alpha\right)\right).$$

Recall that $\mathcal{P}$ is *full* if every superstep requires a $\Theta(\mu)$-relation. In this case, the simulation of a superstep for cluster C incurs a slowdown merely proportional to the cluster size, which is optimal due to the loss of parallelism. We have:

Corollary 4.4.3 *Any T-time full program $\mathcal{P}$ for D-BSP(v, μ, x^α) can be simulated in optimal time $\Theta(Tv)$ on x^α-HMM.*

4.4.2 General Simulation of D-BSP$(v, \mu, f(x))$ on $f(x)$-HMM

The simulation presented in the preceding subsection crucially relies on the assumption that the program $\mathcal{P}$ being simulated is smooth. Such an assumption was made without loss of generality since the access function $f(x) = x^\alpha$ is such that every D-BSP(v, μ, x^α) program admits an equivalent smooth program with the same asymptotic complexity. This is not true, however, if $f(x)$ is subpolynomial (although still polynomially bounded). In this case $f(x)$ grows so slowly that the introduction of dummy supersteps between adjacent supersteps with non-consecutive labels may increase the complexity of the transformed program by more than a constant factor.

Note that if $\mathcal{P}$ is not smooth, then it is no longer true that adjacent supersteps involve clusters of size differing by at most a factor two, which is a property the simulation algorithm in the previous subsection relies upon when performing context swaps in Step 4. Consequently, we need to modify this step to make it work correctly under the new scenario. Consider a round that simulates a certain superstep s of an arbitrary program $\mathcal{P}$ for a cluster C, and suppose that the subsequent superstep $s+1$ in $\mathcal{P}$ is such that $i_{s+1} < i_s$. Let $\hat{C}$ be the i_{s+1}-cluster that contains C; note that $\hat{C}$ is composed by $b \triangleq 2^{i_s - i_{s+1}} \geq 2$ i_s-clusters, including C, which we denote by $\hat{C}_0, \hat{C}_1, \ldots, \hat{C}_{b-1}$. Suppose that $C = \hat{C}_0$, that is, this is the first round executing Superstep s for processors in $\hat{C}$. The simulation will enforce the property that at the beginning of the round the contexts of all processors in $\hat{C}$ are at the top of memory sorted by processor number (i.e., the topmost contexts are those of the processors in $\hat{C}_0$, followed by those of the processors in $\hat{C}_1$, and so on). At this point the simulation enters a *cycle* consisting of b phases, each phase comprising one or more simulation rounds. In the k-th phase, $0 \leq k < b$, the contexts of the processors in $\hat{C}_k$ are brought to the top of memory, then all supersteps up to Superstep s are simulated for these processors, and finally the contexts of $\hat{C}_k$ are put back to the positions occupied at the beginning of the cycle. An example of the memory movements performed during a cycle is depicted in Figure 4.1.

We remark that smaller cycles can open within a cycle, so the simulation algorithm needs a way to keep track of the nested phases: with reference to the notation above, when the simulation of Superstep s for cluster C is over and $i_{s+1} < i_s$, the algorithm must be capable of determining the index k of the phase that has just been completed. We now show that this fundamental operation can be performed with a constant number of arithmetic calculations by looking at the index j of an arbitrary processor $P_j \in C$. By the numbering of clusters and processors we defined in Section 2.2, the number of the i-cluster that contains P_j is given by the i most significant bits of the binary encoding of j: in other words, we can say that

$$P_j \in C^{(i)}_{\lfloor j/2^{\log v - i}\rfloor}, \text{ for } 0 \leq i \leq \log v.$$

In particular, this property holds for $i = i_s$ and $i = i_{s+1}$. Now, let $\hat{C}$ be the i_{s+1}-cluster that contains C, and recall that $\hat{C}$ contains b i_s-clusters, numbered from 0 to $b-1$. Since b is a power of two, from the above observation it straightforwardly follows that the number k such that $C = \hat{C}_k$ is contained in bits $v - i_s, \ldots, v - i_{s+1}$ of j. Consequently, the required index k of the phase that has just been completed is

$$k = \lfloor j/2^{\log v - i_s} \rfloor \text{ div } 2^{i_s - i_{s+1}}.$$

As a whole, Step 4 of the simulation algorithm presented in the previous subsection must be replaced by the following piece of pseudo-code.

```
4    **if** $i_{s+1} < i_s$ **then**
         Let $\hat{C}$ be the $i_{s+1}$-cluster containing $C$, and let $\hat{C}_0, \ldots, \hat{C}_{2^{i_s - i_{s+1}} - 1}$
         be its component $i_s$-clusters, with $C = \hat{C}_k$
4.1      **if** $k > 0$ **then** swap the contexts of $\hat{C}_k$ with those of $\hat{C}_0$
4.2      **if** $k < 2^{i_s - i_{s+1}} - 1$ **then** swap the contexts of $\hat{C}_0$ with those of $\hat{C}_{k+1}$
```

Theorem 4.4.4 *The generalized simulation algorithm is correct.*

Proof We prove that the same invariants **INV1** and **INV2** stated in the previous subsection hold at the beginning of every round: the correctness of the generalized algorithm follows immediately. The proof is by induction on $m = |G|$, where $G \subseteq \{0, 1, \ldots, \log v\}$ is the set of superstep indices available to the D-BSP program.

The claim is trivial for the basis $m = 1$, because in this case it is $G = \{0\}$. Suppose now that the property holds for m and consider a t-step D-BSP program with $|G| = m + 1$; as in the proof of Theorem 4.4.1 we will only illustrate the case in which the program contains exactly one 0-superstep at the end, since the argument easily extends to the case of multiple 0-supersteps. If $t = 1$ then the claim is straightforward, otherwise the algorithm simulates every i_{t-1}-cluster $\hat{C}_0$, $\hat{C}_1$, …, $\hat{C}_{b-1}$ with $b = 2^{i_{t-1} - i_t} = 2^{i_{t-1}}$ (each of which can be seen as a D-BSP with $|G'| = |G| - 1 = m$ superstep types) in a cycle where each such i_{t-1} cluster is taken in turn to the top of memory and supersteps up to $t-1$ are executed for the processors of the cluster. The invariants hold at the beginning of the k-th phase of the cycle, $0 \le k < b$, because the processors of $\hat{C}_k$ have never been executed before. Moreover, they also hold in every round of the phase because of the inductive hypothesis.

After Substep 4.1 of the round in which the last cluster completes Superstep $t - 1$, the i_{t-1}-clusters occupy their original positions; since $k = b - 1$, the swap in Substep 4.2 is not executed and the invariants hold at the beginning of the next, final round. □

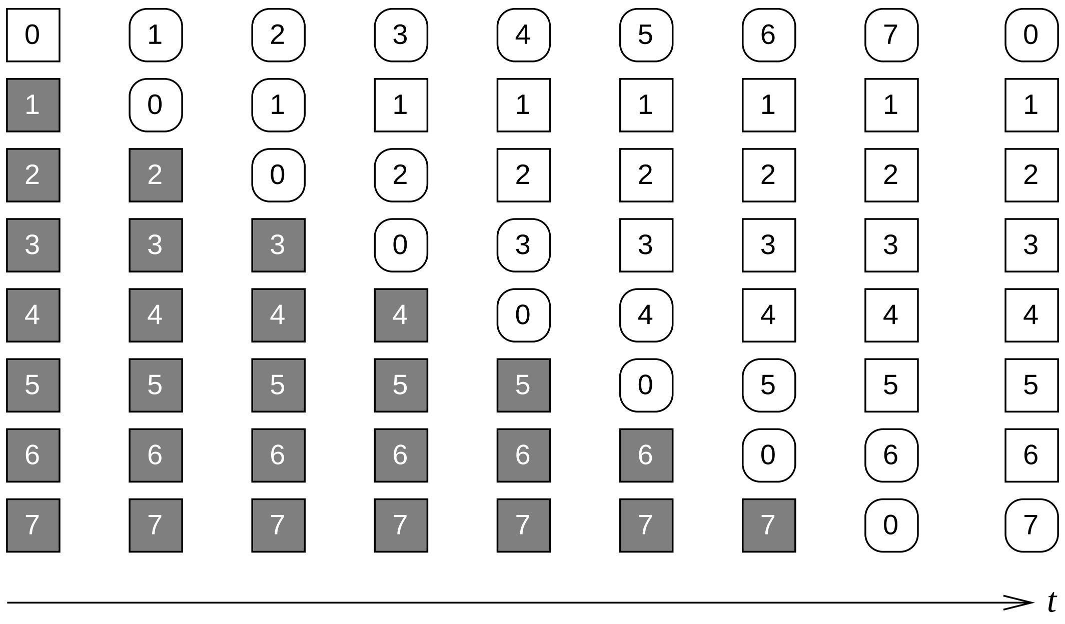

Figure 4.1: snapshots of the HMM memory showing cluster movements during a cycle involving an i_{s+1}-cluster containing $b = 8$ i_s-clusters. Each box represents the processor contexts of a different i_s-cluster. Grey boxes indicate s-ready (i.e., not yet executed) clusters, while white boxes refer to clusters that are being or have already been executed. Snapshots are taken at the beginning of each phase and at the end of the cycle. Rounded boxes indicate the clusters involved in memory swaps during the previous phase.

Since the invariants **INV1** and **INV2** hold, the simulation of a single i_s-cluster during Step 2 can still be performed as we described for the case $f(x) = x^\alpha$. As a consequence, the time for Steps 1, 2 and 3 is dominated by Step 2 and is given by

$$O\left(|C|(\tau_s + \mu f(\mu|C|))\right). \tag{4.1}$$

The memory overhead connected with a cycle (Step 4 of the pseudo-code) needs more careful consideration. Memory blocks $0, 1, |C| - 1$ are accessed a constant number of times for each i_s-cluster that is being simulated, so this cost is dominated by (4.1). Memory blocks $|C|, |C| + 1, \ldots, v/2^{i_{s+1}} - 1$ are accessed twice: the cost $O\left(\mu|C|f(\mu|C|)\right)$ of these accesses is amortized by the cost of the future execution of Superstep $s+1$ for the i_{s+1}-cluster. Finally, the calculation of k pays a negligible cost. We are therefore able to conclude that the overall cost for the simulation of an i_s-cluster is

$$O\left(\frac{v}{2^{i_s}}(\tau_s + \mu f(\mu v/2^{i_s}))\right).$$

By considering the contributions of all the clusters and supersteps in a program, the following results can be straightforwardly proved.

Theorem 4.4.5 *Consider a program $\mathcal{P}$ for D-BSP$(v, \mu, f(x))$ where each processor performs local computation for $O(\tau)$ time, and there are λ_i i-supersteps for $0 \le i \le \log v$. If $f(x)$ is polynomially bounded, then $\mathcal{P}$ can be simulated on $f(x)$-HMM in time*

$$O\left(v\left(\tau + \mu \sum_{i=0}^{\log v} \lambda_i f(\mu v/2^i)\right)\right).$$

Corollary 4.4.6 *If $f(x)$ is polynomially bounded then any T-time full program for D-BSP$(v, \mu, f(x))$ can be simulated in optimal time $\Theta(Tv)$ on $f(x)$-HMM.*

As a concluding remark, we note that the notion of smoothness can be extended to any polynomially bounded function as follows. Let us define $\mathcal{G}(v, \mu, f(x))$ as the set of superstep labels $0 = j_0 < j_1 < \cdots < j_m < \log v$, with the property that for any $0 \le \ell < m$

$$\gamma_1 f(\mu v/2^{j_\ell}) \le f(\mu v/2^{j_{\ell+1}}) \le \gamma_2 f(\mu v/2^{j_\ell}), \tag{4.2}$$

where $0 < \gamma_1 < \gamma_2 < 1$ are suitable constants. We call *$\mathcal{G}$-smooth* a program $\mathcal{P}$ for D-BSP$(v, \mu, f(x))$ if the label of every superstep of $\mathcal{P}$ belongs to $\mathcal{G}(v, \mu, f(x))$, and if for any pair of adjacent supersteps with labels j_ℓ and $j_{\ell'}$ we have $|\ell - \ell'| \le 1$. It is easy to see that for any arbitrary D-BSP$(v, \mu, f(x))$ program there exists a functionally equivalent $\mathcal{G}$-smooth program that exhibits the same asymptotic running time. We remark that the set $\mathcal{G}(v, \mu, f(x))$ may contain much less than $\log v$ elements: for example, if $f(x) = \log x$ then $|\mathcal{G}| = \Theta(\log\log v)$. In general, the slower $f(x)$ grows, the less elements $\mathcal{G}$ has. Since any D-BSP program can be made $\mathcal{G}$-smooth, these observations show that, for certain values of $f(x)$, the network hierarchy can have much less than $\log v$ levels without losing expressive power. Informally, we can say that the network hierarchy "flattens" as $f(x)$ becomes faster and faster: this is a hint that exploiting network locality is particularly important if $f(x)$ grows rapidly.

4.4.3 Analogue of Brent's Lemma

The following theorem generalizes the simulation results of the previous subsections providing an analogue of Brent's lemma [Bre74] for parallel and hierarchical computations.

Theorem 4.4.7 *Consider a program $\mathcal{P}$ for D-BSP$(v, \mu, f(x))$, where each processor performs local computation for $O(\tau)$ time, and there are λ_i i-supersteps, for $0 \leq i \leq \log v$. If $f(x)$ is polynomially bounded, then for any $1 \leq v' \leq v$, $\mathcal{P}$ can be simulated on D-BSP$(v', \mu v/v', f(x))$ in time*

$$O\left(\frac{v}{v'}\left(\tau + \mu \sum_{i=0}^{\log v} \lambda_i f(\mu v/2^i)\right)\right).$$

Proof Let us refer to the D-BSP$(v, \mu, f(x))$ and the D-BSP$(v', \mu v/v', f(x))$ as guest and host machine, respectively. For every $0 \leq j < v'$, the simulation of the processors in cluster $C_j^{(\log v')}$ of the guest machine is assigned to host processor P_j. The memory organization we described in Subsections 4.4.1 and 4.4.2 is used to accommodate the processors in $C_j^{(\log v')}$: in particular, the topmost $\mu v/v'$ memory cells of P_j are organized into v/v' blocks of μ cells each, with block 0 at the top, and block i contains the context of the i-th processor of $C_j^{(\log v')}$. The simulation algorithm also needs constant space per context (which is negligible) to keep track of the simulation state.

For $i < \log v'$, each i-superstep of $\mathcal{P}$ is simulated by two i-supersteps on the host. In the first superstep, P_j takes care of local computation, which is performed by first moving each context to the top of memory; after that, the messages prepared by the v/v' guest processors are copied to the output buffer of P_j and sent to the appropriate destinations. Since the assignment of guest processors to host processors is fixed, the destination of each message can be determined in constant time. In the second superstep, P_j moves the messages it just received to the input buffers of the guest processors. Suppose that the original superstep prescribes local computation for $O(\tau')$ time and the execution of an h-relation: then, the superstep can be simulated in time

$$O\left((v/v')(\tau' + hf(\mu v/2^i)) + \mu f(\mu v/2^i)\right),$$

where the term $(v/v')hf(\mu v/2^i)$ accounts for the execution of an $h(v/v')$-relation within i-clusters on the host. Since $h \le \mu$, this time is dominated by the cost of the memory movements required by the simulation of the v/v' local computations at each host processor; such movements require reading the topmost $\Theta(\mu v/v')$ memory cells no more than a constant number of times, which takes time $O((v/v')\mu f(\mu v/v'))$.

Instead, for $i \ge \log v'$ each i-superstep is simulated sequentially, using the strategy of Subsection 4.4.2: in fact, with the introduction of a final $(\log v')$-superstep (if not already present) any sequence $\mathcal{P}'$ of supersteps whose labels are equal or greater than $\log v'$ can be treated as a program for D-BSP$(v/v', \mu, f(x))$. It is straightforward to see that an i-superstep of $\mathcal{P}$ must be regarded as an $(i - \log v')$-superstep in $\mathcal{P}'$. The slowdown incurred in the simulation of an i-superstep can be obtained by applying Theorem 4.4.5, and is

$$O\left((v/v')\left(\tau' + \mu f\left(\frac{\mu v/v'}{2^{i-\log v'}}\right)\right)\right) = O\left((v/v')\frac{v}{v'}(\tau' + \mu f(\mu v/2^i))\right).$$

The extra time due to the added $(\log v')$-superstep is $O((v/v')\mu f(\mu v/v'))$, and it is amortized by the cost for the simulation of the superstep immediately following $\mathcal{P}'$, which certainly exists since $\mathcal{P}$ ends with a 0-superstep. □

Corollary 4.4.8 *If $f(x)$ is polynomially bounded then any T-time full program for D-BSP$(v, \mu, f(x))$ can be simulated in optimal time $\Theta(Tv/v')$ on D-BSP$(v, \frac{\mu v}{v'}, f(x))$, for any $1 \le v' \le v$.*

4.5 Application to Case Study Problems

In this section we show, on a number of prominent reference problems, how our simulation strategy can be employed to transform efficient D-BSP algorithms into optimal solutions for those problems on the HMM. This provides evidence that the structured parallelism exposed by D-BSP through submachine locality can be automatically transformed into temporal locality on a memory hierarchy. As a consequence, D-BSP can be profitably employed to develop efficient, portable algorithms for hierarchical architectures.

In order to emphasize the transformation of parallelism into temporal locality, for the aforementioned problems we will consider *fine-grained* D-BSP algorithms, i.e.

algorithms where the number of processors v is equal to the problem size n; note that a fine-grained algorithm is also full. In this scenario the size μ of each local memory is constant, hence the D-BSP algorithm is purely based on the exploitation of parallelism and needs not bother with the local hierarchies. Furthermore, for concreteness, we will consider the access functions $f(x) = x^\alpha$, with $0 < \alpha < 1$, and $f(x) = \log x$. Under these functions, upper and lower bounds for our reference problems have been developed directly for the HMM in [AACS87]. We will make use of these HMM results as a comparison stone for the results obtained through our simulation.

Matrix multiplication We call *n-MM* the problem of multiplying two $\sqrt{n} \times \sqrt{n}$ matrices on an n-processor D-BSP using only semiring operations. Both the input matrices and the output matrix are evenly and arbitrarily distributed among the D-BSP processors. We have:

Proposition 4.5.1 *For the n-MM problem there is a D-BSP$(n, 1, x^\alpha)$ algorithm that runs in time*

$$T_{\text{MM}}(n) = \begin{cases} O(n^\alpha) & \text{for } 1/2 < \alpha < 1, \\ O(\sqrt{n}\log n) & \text{for } \alpha = 1/2, \\ O(\sqrt{n}) & \text{for } 0 < \alpha < 1/2, \end{cases}$$

and a D-BSP$(n, 1, \log x)$ algorithm that runs in time $T_{\text{MM}}(n) = O(\sqrt{n})$. The simulation of these algorithms yields optimal performance on the x^α-HMM and the $\log x$-HMM, respectively.

Proof We resort to the standard decomposition of n-MM into eight $(n/4)$-MM subproblems. Let

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

be two square matrices, where A_{ij} B_{ij} denote submatrices of size $n/4$. The problem of calculating $C = A \cdot B$ can be decomposed as follows:

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.$$

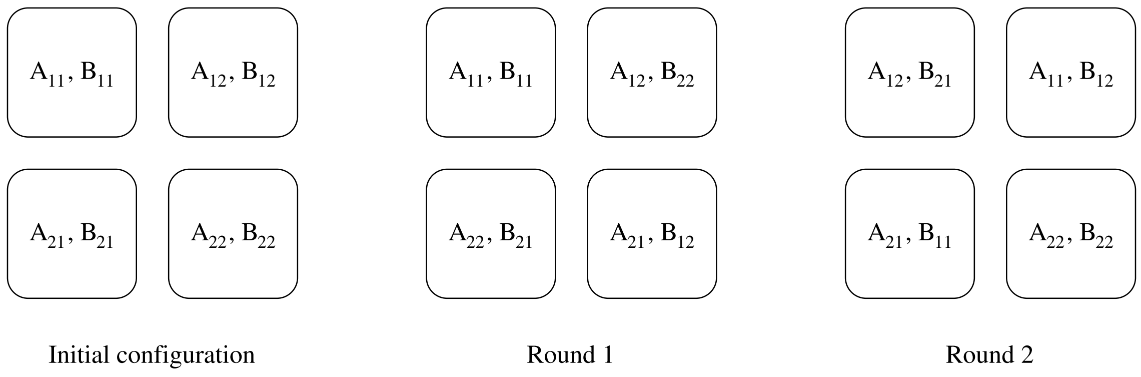

Figure 4.2: assignment of submatrices to the four D-BSP 2-clusters during the execution of the matrix multiplication algorithm.

The eight $(n/4)$-MM subproblems, which are clearly recognizable in the above equation, are recursively solved by the four D-BSP 2-clusters. To keep memory space requirements at a minimum, the subproblems are solved in two rounds; specifically,

1. subproblems $A_{11}B_{11}$, $A_{12}B_{22}$, $A_{22}B_{21}$ and $A_{21}B_{12}$ are solved in Round 1;
2. subproblems $A_{12}B_{21}$, $A_{11}B_{12}$, $A_{21}B_{11}$ and $A_{22}B_{22}$ are solved in Round 2.

Each submatrix is required exactly once in each of the two rounds, hence it is sufficient to have one copy of it; a reasonable assignment of subproblems to the 2-clusters is sketched in Figure 4.2. We remark that data can be moved in the correct position for each round through a constant number of 0-supersteps during which every D-BSP processor exchanges $O(1)$ data: the running time of the algorithm is therefore given by the recurrence equation

$$T_{\mathrm{MM}}(n) = \begin{cases} 2T_{\mathrm{MM}}(n/4) + \Theta(f(n)) & \text{for } n > 1, \\ O(1) & \text{for } n = 1. \end{cases} \tag{4.3}$$

The solution of this equation gives

$$T_{\mathrm{MM}}(n) = O\left(\sqrt{n} + \sum_{i=0}^{(\log n)/2} 2^i f(n/4^i)\right),$$

which leads to the stated running times. (In particular, the solution for $f(x) = \log x$ can be reduced to the solution for the case in which $f(x)$ is a polynomial by observing that $\log x = O(x^\alpha)$, $\alpha < 1/2$.) By Corollary 4.4.6, the simulation of the algorithm on HMM yields a performance matching the lower bound proved in [AACS87]. $\square$

Discrete Fourier Transform We call *n-DFT* the problem of computing the Discrete Fourier Transform of an n-vector evenly and arbitrarily distributed among the n D-BSP processors. We have:

Proposition 4.5.2 *For the n-DFT problem there is a D-BSP$(n, 1, x^\alpha)$ algorithm that runs in time $T^\alpha_{\mathrm{DFT}}(n) = O(n^\alpha)$, and a D-BSP$(n, 1, \log x)$ algorithm that runs in time $T^{\log}_{\mathrm{DFT}}(n) = O(\log n \log\log n)$. The simulation of these algorithms matches the best known bounds for the x^α-HMM and $\log x$-HMM, respectively.*

Proof For D-BSP$(n, 1, x^\alpha)$, we adopt a straightforward schedule of the standard n-input FFT computation dag on the n processors of the D-BSP machine. The algorithm requires $O(1)$ i-supersteps for $0 \le i < \log n$, therefore the running time is

$$O\left(\sum_{i=0}^{\log n - 1} \left(n/2^i\right)^\alpha\right) = O(n^\alpha).$$

Instead, for D-BSP$(n, 1, \log x)$ it is more efficient to resort to the well known algorithm of Cooley and Tukey [CT65]; we quickly remind that the algorithm recursively decomposes an n-DFT problem, with $n = n_1 n_2$, into n_1 independent n_2-DFT problems followed by n_2 n_1-DFT problems. By setting $n_1 = n_2 = n^{1/2}$, we obtain a recursive decomposition of the FFT dag into 2 layers of $\sqrt{n}$ independent $\sqrt{n}$-FFT subgraphs; the two layers are separated by a permutation, i.e. by a 0-superstep. Each subgraph can be scheduled onto a distinct D-BSP submachine with $\sqrt{n}$ processors, hence the running time of the Cooley-Tukey algorithm on D-BSP$(n, 1, f(x))$ is given by the recurrence equation

$$T^{\log}_{\mathrm{DFT}}(n) = \begin{cases} 2T^{\log}_{\mathrm{DFT}}(\sqrt{n}) + O(f(n)) & \text{for } n > 1, \\ O(1) & \text{for } n = 1. \end{cases}$$

By solving the recurrence for $f(x) = \log x$, we have:

$$T^{\log}_{\mathrm{DFT}}(n) = O\left(\sum_{i=0}^{\log\log n - 1} 2^i \log\left(n^{1/2^i}\right)\right) = O(\log n \log\log n).$$

The performance of the HMM algorithms obtained by applying the results stated in Corollaries 4.4.3 and 4.4.6 matches the best known bounds of $O\left(n^{1+\alpha}\right)$ on the x^α-HMM and of $O\left(n \log n \log\log n\right)$ on the $\log x$-HMM, proved in [AACS87]. □

Sorting We call *n-sorting* the problem in which n keys are initially evenly distributed among the n D-BSP processors and have to be redistributed so that the smallest key is held by processor P_0, the second smallest one by processor P_1, and so on. We have:

Proposition 4.5.3 *There is an n-sorting algorithm for D-BSP$(n, 1, x^\alpha)$ that runs in time*

$$T^{\alpha}_{\mathrm{SORT}}(n) = O\left(n^\alpha\right).$$

The simulation of this algorithm on x^α-HMM exhibits optimal performance.

Proof Proposition 2.4.2 gives a sorting algorithm that runs in time $T_{\mathrm{SORT}}(1, n) = O\left(n^\alpha\right)$ on D-BSP$(n, 1, x^\alpha)$; by Corollary 4.4.3, this algorithm can be simulated in time $O\left(n^{1+\alpha}\right)$ on x^α-HMM. Optimality follows from the lower bound proved in [AACS87]. □

We remark that all n-sorting strategies known in the literature for BSP-like models seem to yield $\Omega\left(\log^2 n\right)$-time algorithms when implemented as fine-grained on D-BSP$(n, 1, \log x)$. By simulating one such algorithm on the $\log x$-HMM we get a running time of $\Omega\left(n \log^2 n\right)$, which is a $\log n / \log\log n$ factor away from optimal [AACS87]. However, such non-optimality is due to the inefficiency of the D-BSP algorithm employed and not to a weakness of the simulation. In fact, our simulation implies a $\Omega\left(\log n \log\log n\right)$ lower bound for n-sorting on D-BSP$(n, 1, \log x)$. No tighter lower bound or better algorithm are known so far.

Chapter 5

Parallelism and Spatial Locality

The design of algorithms exhibiting a high degree of temporal and spatial locality of reference is crucial to attain good performance on current and foreseeable computing systems featuring ever deeper memory hierarchies. Previous work has demonstrated that task parallelism can be efficiently transformed into locality of reference in two-level hierarchies; in Chapter 4, we moved a step forward and showed how the more structured type of parallelism exposed by submachine locality can be efficiently turned into temporal locality on arbitrarily deep hierarchies. In this chapter, we complete and extend the above result by encompassing both temporal and spatial locality. Specifically, we present two schemes to simulate parallel algorithms designed for an extension of D-BSP that models locality of reference: the first scheme attains a slowdown proportional to the loss of parallelism, while the second one, under certain conditions, is able to yield sequential algorithms for the Hierarchical Memory Model with Block Transfer [ACS87] with a slowdown *less than proportional* to the loss of parallelism. The resulting simulations give good hierarchy-conscious sequential algorithms from parallel ones, and provide evidence of the strict relation between submachine locality in parallel computation and locality of reference (both temporal and spatial) in the hierarchical memory setting. Crucial to these results is the block transfer mechanism of the BT model, which sets no upper limit on the size of the block that can be transferred at once. We will name such a mechanism *unlimited block transfer*. Unlimited block transfer is in contrast with the considerable body of work on RAM-disk memory hierarchies started in [VS94a], but has been advocated in [ACS89, BEP02].

The remainder of the chapter is organized as follows. Section 5.1 is a survey on existing results concerning parallelism and spatial locality. Section 5.2 sketches our original contributions on the subject. In Section 5.3 we describe how to include block transfer in the D-BSP model. Then, in Section 5.4, on such an augmented model we develop an evolutionary simulation scheme that is directly inspired by the one of Chapter 4 and naturally leads to a Brent-like lemma for block transfer, with slowdown proportional to the loss of parallelism. After that, in Section 5.5 we introduce a new scheme that makes better use of spatial locality; indeed, the block transfer mechanism is exploited so efficiently that the cost of simulating a superstep does not depend on the access cost function $f(x)$ any more, hence codes tuned for very different machines yield the same running time once simulated. As a corollary of this fact, we show that fine-grained algorithms developed for a D-BSP with logarithmic cost function are simulated efficiently on a whole class of memory hierarchies. An extensive comparison of the two simulation schemes is performed in Section 5.6, which discusses several implications of our results and their application to some relevant case studies.

5.1 Previous Work

In the last decade, a number of computational models have been proposed which explicitly account for the hierarchical structure of the memory system. The following models can be considered relevant for our study.

The *Hierarchical Memory Model* (HMM), introduced in [AACS87] and defined in Section 4.3, is a random access machine where access to memory location x requires time $f(x)$, thus encouraging the exploitation of temporal locality. The *Hierarchical Memory Model with Block Transfer* (BT, for short) [ACS87, VS94b] was subsequently introduced with the intention of also rewarding spatial locality by augmenting HMM with the capability of moving blocks of memory at a reduced cost.

The P-HMBT model [JW94] is a variant of BT which changes the memory copy mechanism. Memory is divided into levels, with cells in the same level exhibiting the same access cost, and block transfers can only take place between adjacent levels; copy operations involving distinct levels can take place in parallel. The model is more powerful than BT in some situations (e.g., the touching problem), but sometimes it is weaker (e.g., search operations) at least for the $\log x$ cost function. The Pipelined P-HMBT, or PP-HMBT for short [JW94], is an empowered P-HMBT where multiple block transfers between adjacent levels can take place simultaneously; anyway, only one word can be transferred between adjacent levels in a "clock" cycle. This model is strictly more powerful than BT.

The Block Move (BM) model [Reg96] is an extension of BT that allows memory pipelining not only for memory copy operations, but for a whole set $\mathcal{T}$ of complex primitives called *transductions*. A transduction $t \in \mathcal{T}$ maps the content of a memory block $[x, x+l]$ into a disjoint, output block $[y, y+l]$ of the same size. In general, the content of block $[x, x+l]$ is altered by t before it is written to the output block: the transformation operated by t is described by a *deterministic generalized sequential machine*, which is a sort of finite state automata. Not surprisingly, BM is strictly more powerful than BT.

In the LPM model [LP93], the cost of an access to a size-m memory is $\log m$ regardless of the memory address; this flat cost model makes it possible to pipeline an unlimited number of memory requests in an easy and nice way. The memory subsystem accepts new requests at a fixed rate; the requests are processed in the order in which they are submitted. An LPM algorithm is at most $\log m$ times slower than its RAM counterpart (in the worst case, the algorithm exhibits no spatial locality, hence it uses no pipelining); under the reasonable hypothesis that m is polynomial in the input size n, the "LPM slowdown" is at most logarithmic.

Finally, a two-level memory organization is featured by the *External Memory* (EM) model [VS94a], which has been extensively used in the literature to develop I/O-efficient algorithms. The model provides a finite-size internal memory connected to multiple disks; during an I/O operation, it is possible to simultaneously read/write B consecutive records from each of the disks.

Few parallel models have dealt with spatial locality in the network hierarchy; the most relevant of them are BSP* [BDMadH98, BDP99], which we already introduced in Section 2.1, and BPRAM [ACS89]. Remember that BSP* is a BSP variant which sets the cost of communication during a superstep to $g_1 b + gh + \ell$, where $gh + \ell$ is the standard BSP cost function and b is the maximum *aggregate* size of the messages exchanged by a processor. In the BPRAM model, every processing node is endowed with a local, flat-cost memory of unlimited size; the nodes are then connected to a shared memory module that is used for message exchange. A block of b consecutive cells can be transferred from the shared memory in time $b+\ell$, where ℓ is a fixed start-up cost. With a generalized cost function of $g_1 b + \ell$, which accounts for bandwidth limitations, BPRAM yields good performance predictions in the experimental study of Juurlink and Wijshoff [JW98]. We remark that no parallel model in the literature deals with spatial locality in both the network and the memory hierarchy.

Earlier works provided evidence that efficient sequential algorithms for two-level hierarchies can be obtained by simulating parallel ones. In [DDH03, DDHM99, SK97] schemes are presented that simulate parallel algorithms designed for coarse-grained parallel models, such as BSP [Val90a], BSP* [BDMadH98], and CGM [DFRC96], on the EM model. The main intuition behind these works is that the interleaving between large local computation and bulk communication phases, which characterizes coarse-grained parallel algorithms, maps nicely on the two-level structure of the EM model. However, the flat parallelism offered by the above coarse-grained models is unable to afford the finer exploitation of locality which is required by deeper hierarchies.

Another approach to the development of efficient EM algorithms was proposed in [CGG$^+$95] based on the simulation of fine-grained PRAM algorithms. Beside proving a general simulation result, the authors show how to turn PRAM computations that involve geometrically smaller subsets of processors into highly efficient EM algorithms. This suggests that some form of submachine locality in the parallel setting can be profitably transformed into locality of reference in the memory accesses. The simulation of PRAM algorithms to obtain efficient sequential ones has also been explored in [Vis96], where parallelism is turned into efficient cache prefetching strategies.

5.2 Original Contributions

The objective of this chapter is to continue the investigation of Chapter 4 to encompass temporal *and* spatial locality, so to assess to what extent structured parallelism can lead to a combined exploitation of both forms of locality of reference in multi-level hierarchies. More specifically, we first extend D-BSP to model spatial locality in both memory and interconnection network; then, building on the results of the previous chapter, we devise two strategies to simulate D-BSP algorithms on a host machine exhibiting a reduced degree of parallelism. The first strategy shows that a Brent-like lemma can still be attained in the new, broader scenario. The second strategy reveals that efficiency in the BT model can be achieved starting from D-BSP algorithms exhibiting a much coarser level of submachine locality than the one needed by the simulation on HMM, which required a decomposition into submachines strictly dependent on the access cost function. However, some examples provide evidence that a certain level of submachine locality must nonetheless be exhibited by the D-BSP algorithm in order to achieve optimal or quasi-optimal BT algorithms. Our results are in accordance with the ones in [ACS87], which show that an efficient exploitation of the powerful block transfer capability of the BT model is able to hide access costs almost completely.

The importance of our contribution is twofold. On the one hand, to the best of our knowledge, ours is the first work that establishes a relation between the network locality embodied in parallel algorithms and both temporal and spatial locality of reference in sequential algorithms for general hierarchies. On the other, our simulation provides a tool to obtain efficient BT algorithms automatically from the large body of parallel algorithms developed in the literature over the last two decades.

5.3 Machine Models

5.3.1 BT

The $f(x)$-BT (Hierarchical Memory Model with Block Transfer) was introduced in [ACS87] by augmenting the $f(x)$-HMM model of [AACS87] with a block transfer facility. Specifically, as in the HMM, an access to memory location x requires time $f(x)$, for a given non-decreasing function $f(\cdot)$, but the model makes it also possible to copy a block of l memory cells $[x-l+1, x]$ into a *disjoint* block $[y-l+1, y]$ in time $\max\{f(x), f(y)\} + l$, for arbitrary $l > 1$.

It must be remarked that the block transfer mechanism featured by the model allows memory pipelining only for copy operations and only for non-overlapping memory regions; such mechanism is nevertheless powerful since it allows for the pipelined movement of arbitrarily large blocks. This is particularly noticeable if we look at the fundamental *touching* problem, which requires to bring each of a set of n memory cells to the top of memory. The following proposition is proved in [ACS87].

Proposition 5.3.1 *The touching problem on $f(x)$-BT requires time $\Theta(n \log^* n)$ if $f(x) = \log x$, and $\Theta(n \log\log n)$ if $f(x) = x^\alpha$, for a positive constant $\alpha < 1$.*

This proposition gives a nontrivial lower bound on the execution time of many problems where all the inputs, or at least a constant fraction of them, must be examined. For the sake of comparison, observe that on $f(x)$-HMM the touching problem requires time $\Theta(nf(n))$, which shows the added power introduced by block transfer.

Although the powerful transfer capability of BT might appear unrealistic with respect to current technology, the architectural feasibility of unlimited pipelined transfers within memory hierarchies has recently been advocated in [BEP02].

5.3.2 D-BSP with Block Transfer

In this subsection we augment the D-BSP model with block transfer. Our aim is to extend the model we presented in Chapters 2 and 4 by adding block transfer facilities to both the memory hierarchy and the network hierarchy; still moving on the "seamless integration" path of Chapter 4, we want these two facilities to be alike and equally powerful. Moreover, although we are introducing yet another model of computation, we intend to resort as much as possible to features for block transfer that are already established in the literature. These guidelines have been met as follows, giving birth to what we call the D-BSPB model.

- For the memory hierarchy inside each processing node, block transfer is shaped after the mechanism of $f(x)$-BT: the cost of a single access to memory cell x is $f(x)$, and it is possible to copy an interval of memory cells $[x-l+1, x]$ into a disjoint interval $[y-l+1, y]$ in time $\max\{f(x), f(y)\}+l$.

- For the network hierarchy, block transfer is modeled as proposed in [BDP99] for the BSP* model [BDMadH98]: an i-superstep is charged a cost

$$\tau + b + f(\mu v/2^i) \cdot h.$$

  As in Chapter 4, μ is the size of the local memories, τ is the maximum time spent by a processor for local operations and h is the maximum number of messages that a processor exchanges. The new variable is b, the maximum aggregate size (in machine words) of the messages exchanged by a processor.

The two points above provide a complete description of the mechanisms with which the D-BSPB model deals with temporal, spatial and network locality; in all other respects, a D-BSPB behaves exactly as the standard D-BSP we have already described. In particular, note that D-BSPB is still specified by three parameters: the number v of (virtual) processors, the size μ of the local memory available to each processor, and the cost function $f(x)$. As a consequence, in what follows we will indicate a D-BSP with Block Transfer as D-BSPB $(v, \mu, f(x))$. Unless otherwise noted, $f(x)$ can be an arbitrary function satisfying Definition 4.3.1. The D-BSPB model is an extension of D-BSP, in the sense that a T-time D-BSP$(v, \mu, f(x))$ program (i.e., a program that makes no use of block transfer) can be executed in time $O(T)$ on D-BSPB $(v, \mu, f(x))$. The converse is not true, since D-BSPB $(v, \mu, f(x))$ may be strictly more powerful than D-BSP$(v, \mu, f(x))$. Consider the trivial example that follows:

- a D-BSPB $(1, \mu, x^\alpha)$ can sort n items from its internal memory in $O(n \log n)$ time [ACS87], whereas a D-BSP$(1, \mu, n^\alpha)$ cannot;

- each processor of a D-BSPB (v, μ, x^α) can send a constant fraction of its local memory to a distinct processor (permutation) in time $O(\mu)$ if μ is big enough, whereas a D-BSP(v, μ, n^α) cannot.

As we have already noticed for the BT model, the block transfer mechanism featured by D-BSPB is rather powerful since it offers unlimited block transfer: in fact, it is possible to move arbitrary blocks of memory (a sort of DMA transfer) or arbitrarily large network messages in a single operation. This is not true in current implementations of DRAM memories (which only allow the pipelined transfer of a small, constant number of memory cells for each memory request) and hard disks (where the block size is fixed). Moreover, internal memory is shaped on BT memory and consequently exhibits unitary *injection ratio*, i.e. it is able to accept a new pipelined memory access in each clock cycle. Since we want memory pipelining and network pipelining to be equally powerful, we must force the D-BSPB network to exhibit unitary injection ratio as well: once the start-up cost $f(\mu v/2^i)$ has been paid for a message, the payload is transferred on the network at the rate of one word per cycle. All in all, this means that bandwidth limitations are completely neglected in our model, or, better, that hardware pipelining is assumed to be so powerful to hide them completely. This is acceptable because, in this chapter, our study is focused on spatial locality: we want to see what advantages can be reaped from block transfer in its most powerful form. In a broader scenario, bandwidth can be readily added to the model by making access times to memory and network dependent on injection ratio parameters; this has already been done, for example, in the $(f(x), r)$-PHM memory architecture [BEP02]. A reasonable way to modify the cost function of D-BSP is the following.

- The cost of copying memory region $[x-l+1, x]$ into the disjoint region $[y-l+1, y]$ is set to $\max\{f(x), f(y)\} + g_{-1}l$.

- The cost of performing an h-relation inside an i-cluster, $0 \leq i \leq \log v$, is set to $g_i b + hf(\mu v/2^i)$, with b the maximum number of words exchanged by a processor.

In this fashion, bandwidth for the memory and the network hierarchies is given by a vector $\boldsymbol{g} = (g_{-1}, g_0, \ldots, g_{\log v})$ of length $\log v + 1$. The D-BSPB model we use in this chapter can clearly be obtained by taking $\boldsymbol{g} = (1, 1, \ldots, 1)$.

5.4 The Plain Simulation Scheme

In this section we discuss a scheme to simulate a D-BSPB $(v, \mu, f(x))$ program $\mathcal{P}$ on D-BSPB $(v', \mu v/v', f(x))$. We move from the simulation scheme of Section 4.4.2 and we show that, by modifying it to introduce block transfer whenever possible, we still obtain a slowdown that is proportional to the loss of parallelism. This result is not surprising, but it is not completely trivial either: D-BSPB $(v, \mu, f(x))$ is more powerful than D-BSP$(v, \mu, f(x))$ as a parallel model, hence the loss of parallelism potentially brings a higher slowdown in the first model than in the second one. Our results provide evidence that this is not the case, thanks to the availability of block transfer in the memory hierarchy.

We need not prove the correctness of the modified scheme because it performs the same choices and uses the same data structures as the one in Section 4.4.2; as a consequence, we focus on the analysis of execution time. We incidentally remark that the notion of $\mathcal{G}$-smoothness introduced in Subsection 4.4.2 can be extended to D-BSPB $(v, \mu, f(x))$ without modifications; moreover, any D-BSPB program can be made $\mathcal{G}(v, \mu, f(x))$-smooth without asymptotically increasing its running time.

Subsection 5.4.1 considers the case in which the simulating (guest) machine is sequential ($v' = 1$): in this case, the D-BSPB model reduces to the BT model. Then, Subsection 5.4.2 deals with the general case in which there is only a partial loss of parallelism.

5.4.1 Simulation of D-BSPB $(v, \mu, f(x))$ on $f(x)$-BT

Although the simulation scheme of Section 4.4.2 gives a correct algorithm for the BT model, it is clearly doomed to give a poor execution time since it makes no use of spatial locality. Therefore, it is necessary to re-examine every step of the scheme to see if its efficiency can be increased by means of block transfer. We can pinpoint two opportunities for improvement: swaps of contexts and message delivery.

The first opportunity for improvement is given by all the steps that prescribe a swap of processor contexts (Steps 2.1 and 2.3) or even groups of $|C| \geq 1$ processor contexts (Steps 4.1 and 4.2): these memory regions are at least μ cells in size, so they are clearly eligible for block transfer. However, there is a technical difficulty to face. The BT model does not allow to swap two memory blocks in place: since the only block transfer operation available is the copy of disjoint memory blocks, in order to implement a swap we need some buffer space, that is, extra space where data can be freely copied without overwriting processor contexts or data structures. Such a buffer must be as small as possible and it must reside in a fast memory region. To keep the swaps efficient, it is enough to create an extra swap buffer of size μ and place it into memory cells $\mu, \mu+1, \ldots, 2\mu-1$: memory blocks numbered from 1 to $v-1$ are pushed down accordingly, hence a memory cell $x \geq \mu$ is moved to cell $x+\mu$. After the introduction of the buffer, the amount of memory required by the simulation scheme is $O(\mu v + \mu) = O(\mu v)$. Memory block 0 does not change its position, so the access time to its cells is unchanged; for $x \geq \mu$, the time to access cell x is increased from $f(x)$ to $f(x+\mu) \leq f(2x) = \Theta(f(x))$, since $f(x)$ is polynomially bounded. As a consequence, the introduction of the swap buffer does not asymptotically alter the amount of memory for the simulation or the access time to the cells of any memory block, so the presence of the buffer can be neglected during all operations except memory swaps. With regard to this point, swapping the contents of two memory regions A and B relies on the buffer as described by the trivial pseudo-code that follows.

```
        $k \leftarrow \lceil |A|/\mu \rceil$
        **for** $i \leftarrow 0$ **to** $k-1$ **do**
S.1         Copy cells $i\mu, \ldots \min\{|A|, (i+1)\mu\} - 1$ of region $A$
            into the buffer
S.2         Copy cells $i\mu, \ldots \min\{|A|, (i+1)\mu\} - 1$ of region $B$
            into the corresponding cells of region $A$
S.2         Copy the content of the buffer into cells
            $i\mu, \ldots \min\{|A|, (i+1)\mu\} - 1$ of region $B$
```

According to this pseudo-code, straightforward calculations lead us to conclude that the contents of memory blocks 0 and j (Steps 2.1 and 2.3 of the simulation scheme) can be swapped by the guest $f(x)$-BT machine in time

$$O(f(j\mu) + \mu). \tag{5.1}$$

Furthermore, the $|C|$ blocks on top of memory can be swapped with the k-th group of $|C|$ blocks (Steps 4.1 and 4.2 of the scheme) in time

$$O\left(\sum_{j=0}^{|C|-1} \left(f\left((k|C|+j)\mu\right)+\mu\right)\right). \tag{5.2}$$

The second opportunity to exploit block transfer arises in the simulation of message exchanges (Step 2.4 of the simulation scheme). To be more specific, every message can be seen as a sequence of consecutive memory locations that can be copied to the appropriate input buffer by means of block transfer. The advantage of such a strategy depends on the number and size of the messages: the longer a message, the more effective the memory copy. Anyway, if the simulated program is efficient we expect block transfer to be useful on average, since the D-BSPB $(v, \mu, f(x))$ model rewards the use of long messages. To estimate the cost of performing the message exchange prescribed for Superstep s of cluster C, let us first define the following quantities.

- h_k, $1 \le k \le |C|$: number of messages sent by the k-th processor of cluster C.
- r_k, $1 \le k \le |C|$: number of messages received by the k-th processor of C.
- b_l^k, $1 \le l \le h_k$: length of the l-th message sent by processor k.

By using these quantities, the time required for the message exchange can be expressed as

$$O\left(\sum_{k=1}^{|C|} r_k f(k\mu) + \sum_{k=1}^{|C|}\sum_{l=1}^{h_k}\left(f(k\mu-l)+b_l^k\right)\right),$$

where the first term is the time for pointer updating and the second one is the time to actually copy each message to the appropriate input buffer. Since $h_k \le h$, $r_k \le h$, and $\sum_k \sum_l b_l^k \le |C|b$, this expression can be immediately rewritten as

$$O\left(h\sum_{k=1}^{|C|} f(k\mu) + |C|b + \sum_{k=1}^{|C|}\sum_{l=1}^{h} f(k\mu-l)\right)$$

and then bounded from above as

$$O\left(h\sum_{k=1}^{|C|} f(k\mu) + |C|b\right). \tag{5.3}$$

The analysis of the two improvements is complete; it is now possible to calculate the time required by the simulation algorithm. Remember that, by our analysis in Subsections 4.4.1 and 4.4.2, such a time can be split into two main components: the cost for the simulation of one superstep for the topmost cluster C, and the cost of selecting a new cluster when the superstep is over. The first component includes:

1. the cost for the simulation of local computation, which is the sum of the computation times of the $|C|$ processors in the cluster;
2. the cost for message exchange, which is now given by Equation (5.3);
3. the cost of copying each context of the cluster into the topmost memory block, which is obtained by summing the quantity (5.1) for all values of j between 1 and $|C|$.

As a whole, the cost for the simulation of Superstep s for cluster C is

$$O\left(|C|\tau_s + |C|\mu + |C|b + h\sum_{j=1}^{|C|} f(j\mu)\right) = O\left(|C|\left(\tau + \mu + b + hf(\mu|C|)\right)\right),$$

where $|C| = v/2^{i_s}$. The second component of the cost, which is present only if $i_{s+1} < i_s$, is due to the swap of the $|C|$ topmost blocks with a new set of blocks, as bounded by Equation (5.2). Recall that, when the processors in the i_{s+1}-cluster are ready to begin Superstep i_{s+1}, k has been assigned all the integer values from 0 to $2^{i_s - i_{s+1}} - 1$ inclusive. By inspecting Equation (5.2) in the light of this observation, it is easy to infer that the overall time for swaps is asymptotically equal to the time for reading the topmost $v/2^{i_{s+1}}$ blocks a constant number of times; as usual, the cost of these accesses is amortized by the cost of the future execution of Superstep $s+1$ for the i_{s+1}-cluster. By combining these observations and remembering that $h \leq b \leq \mu$, we conclude:

Theorem 5.4.1 *Consider a program $\mathcal{P}$ for D-BSPB $(v, \mu, f(x))$ where each processor performs local computation for $O(\tau)$ time, and there are λ supersteps. If $f(x)$ is polynomially bounded, then $\mathcal{P}$ can be simulated on $f(x)$-BT in time*

$$O\left(v\left(\tau + \sum_{s=0}^{\lambda-1}\left(h_s f(\mu v/2^{i_s}) + \mu\right)\right)\right),$$

where h_s is the degree of the relation prescribed by Superstep s.

Note that, since $h_s \leq \mu$, we can still say that the simulation time is

$$O\left(v\left(\tau + \mu \sum_{i=0}^{\log v} \lambda_i f(\mu v/2^i)\right)\right)$$

as we did in Theorem 4.4.5, but this is too rough an approximation for D-BSPB.

A D-BSPB program $\mathcal{P}$ is *full* if $b = \Theta(\mu)$ in every superstep. It must be noticed that this definition is different from the one of Chapter 4, which bounded the degree h of the relation in every superstep. However, the spirit of the definition remains the same: in a full program, each processor sends/receives an amount of data proportional to its local memory size. We have:

Corollary 5.4.2 *If $f(x)$ is polynomially bounded then any T-time full program for D-BSPB $(v, \mu, f(x))$ can be simulated in time $\Theta(Tv)$ on $f(x)$-BT.*

5.4.2 Analogue of Brent's Lemma

This subsection describes a general simulation strategy that is capable of simulating a D-BSPB $(v, \mu, f(x))$ program $\mathcal{P}$ on D-BSPB $(v', \mu v/v', f(x))$, with $1 \leq v' \leq v$; the corresponding simulation time shows that a Brent-like lemma still holds for the D-BSPB model. Since a T-time D-BSP$(v, \mu, f(x))$ program is also a $O(T)$-time D-BSPB $(v, \mu, f(x))$ program, this result is a natural extension of the one of Subsection 4.4.3.

The idea behind the simulation strategy is identical to the one that is described in the proof of Theorem 4.4.3. For every $0 \leq j < v'$, the processors in cluster $C_j^{(\log v')}$ of the guest machine are assigned to host processor P_j. For $i < \log v'$, each i-superstep of $\mathcal{P}$ is simulated by an i-superstep on the host; the simulation takes time

$$O\left((v/v')(\tau + hf(\mu v/2^i) + \mu)\right), \tag{5.4}$$

where τ is the maximum time spent by a guest processor to perform local computation. The term $(v/v')(hf(\mu v/2^i) + \mu)$ accounts for the execution of an (hv/v')-relation within i-clusters and dominates the time $O((v/v')(f(\mu v/v') + \mu))$ required by memory movements during the simulation of the v/v' local computations at each host processor.

Instead, for $i \geq \log v'$, each i-superstep is simulated sequentially, using the strategy of Subsection 4.4.2 with the improvements of Subsection 5.4.1. The slowdown incurred in the simulation of any of these supersteps is obtained by applying Theorem 5.4.1 while setting the number of processors of the guest machine to v/v', and is still given by Equation (5.4).

If $\mathcal{P}$ has λ supersteps, and the s-th superstep, $0 \leq s < \lambda$, prescribes an h_s-relation, then, by summing up the contributions (5.4) for all the λ supersteps, the time required for the simulation of $\mathcal{P}$ turns out to be

$$O\left(\frac{v}{v'}\left(\tau + \sum_{s=0}^{\lambda-1}\left(h_s f(\mu v/2^{i_s}) + \mu\right)\right)\right). \tag{5.5}$$

The following corollary is an immediate application of Equation (5.5) to the special case in which $\mathcal{P}$ is full.

Corollary 5.4.3 *If $f(x)$ is polynomially bounded then any T-time full program for D-BSPB $(v, \mu, f(x))$ can be simulated in time $\Theta(Tv/v')$ on D-BSPB $(v, \frac{\mu v}{v'}, f(x))$, for any $1 \leq v' \leq v$.*

5.5 An Advanced Simulation Scheme

In this section we present an advanced scheme to simulate a D-BSPB $(v, \mu, f(x))$ program $\mathcal{P}$ on $f(x)$-BT. We will refer to D-BSP and BT as the guest and host machine, respectively. We assume that $f(x) = O(x^\alpha)$, for some arbitrary constant $0 \leq \alpha < 1$, and that $\Theta(v \log\log v)$ memory is available on the host BT machine. Note that all relevant BT access functions $f(x)$ considered in the literature [AACS87, ACS87] are captured by the above scenario.

In Subsection 5.5.1, we explain the main idea behind the new scheme and the corresponding memory organization; then, in Subsection 5.5.2 we describe and analyze the scheme. Subsection 5.5.3 presents some observations for the case in which the host machine has $v' > 1$ processors.

5.5.1 Memory Organization

If we look at the simulation scheme of Section 5.4, we see that it makes only a limited use of block transfer; for example, the scheme reads one context at a time during the simulation of local computations. This is usually inefficient because the cost $f(\mu j)$ paid when reading the j-th context of the cluster under simulation may not be amortized by the size μ of the context itself: in a broad sense, the start-up cost is not compensated by a corresponding "data processing" activity on a sufficient number of memory cells. The correct mode of operation is suggested in [ACS87]: a good BT algorithm must be recursive, and block transfer must be used at every level of recursion. For our simulation scheme, we interpret this rule as the need to simulate an i-cluster by recursively reading chunks of approximately $f(\mu v/2^i)$ memory cells: this strategy ensures an optimal balance between the start-up cost that is paid to access the chunk and the cost of reading the chunk by means of a single block transfer operation. This simulation strategy gives birth to the COMPUTE subroutine, whose pseudo-code will be given in Subsection 5.5.2; for the time being, to understand the memory organization it is enough to know that the chunk size for recursive calls is decided by function $c(n, m)$ defined below.

Definition 5.5.1 *Let n, m be two positive integers; $c(n, m)$ is the greatest power of 2 such that $c(n, m) \leq \min\{f(nm)/m, n/2\}$.*

A straightforward consequence of this definition is that

$$\frac{1}{2}\min\{f(nm)/m, n/2\} < c(n, m) \leq \min\{f(nm)/m, n/2\}. \tag{5.6}$$

If COMPUTE is invoked on an i-cluster C, then the chunk contains $c(|C|, \mu) = c(v/2^i, \mu)$ processor contexts. Note that the size of the chunk is chosen in a way that does not split any context into multiple parts, a property which greatly simplifies the description of the simulation.

As far as chunks are concerned, it is necessary to face a technical problem: as we have already said, copying a memory block of size $c(n, \mu)$ in one operation requires a buffer of size $c(n, \mu)$. The buffer cannot be too far from the block for efficiency reasons: for example, if the buffer were simply placed after all the v processors' contexts, then the time to access the buffer would be high enough to rise the cost of the entire simulation. As a consequence, the required buffer space must be interspersed with the contexts. During the simulation, buffer space is rearranged as needed by means of PACK and UNPACK subroutines. More specifically, UNPACK(i), with $0 \leq i \leq \log v$, is invoked when all contexts of an i-cluster are consecutively stored on top of memory, followed by an empty space equal to the cluster size (i.e., $v/2^i$ empty blocks). The code for UNPACK(i) is the following:

```
UNPACK($i$)
    **if** $i = \log v$ **then exit**
    Shift blocks $v/2^{i+1}, \ldots, v/2^i - 1$ to blocks $v/2^i, \ldots, 3v/2^{i+1} - 1$
    {creates buffer space to simulate the topmost $(i+1)$-cluster}
    UNPACK($i+1$)
```

The net effect of a call to UNPACK(i) when an i-cluster C is on top of memory, is to intersperse the $v/2^i$ empty blocks which followed C among the contexts of C itself. Figure 5.1 illustrates how the memory layout is modified by a call to UNPACK(0). The subroutine PACK(i) performs the same operations of UNPACK(i) but in reverse order, thus compacting the contexts belonging to the topmost i-cluster: after a call to PACK(i), the i-cluster is therefore followed by a free memory area having the same memory footprint of the cluster. The pseudo-code of PACK(i) follows.

```
PACK($i$)
    **if** $i = \log v$ **then exit**
    PACK($i+1$)
    Shift blocks $v/2^i, \ldots, 3v/2^{i+1} - 1$ to blocks $v/2^{i+1}, \ldots, v/2^i - 1$
    {expunges buffer space from the topmost $i$-cluster}
```

The buffer management process guarantees that the starting memory address for each context in C is at most doubled by the presence of the buffers. Since $f(x)$ is polynomially bounded, we can conclude that the buffers do not alter memory access time by more than a multiplicative constant.

Since PACK(i) and UNPACK(i) read the same memory cells, the execution time of both subroutines is given by the same expression. The shift operation prescribed by PACK and UNPACK can be performed as a single block transfer by relying on

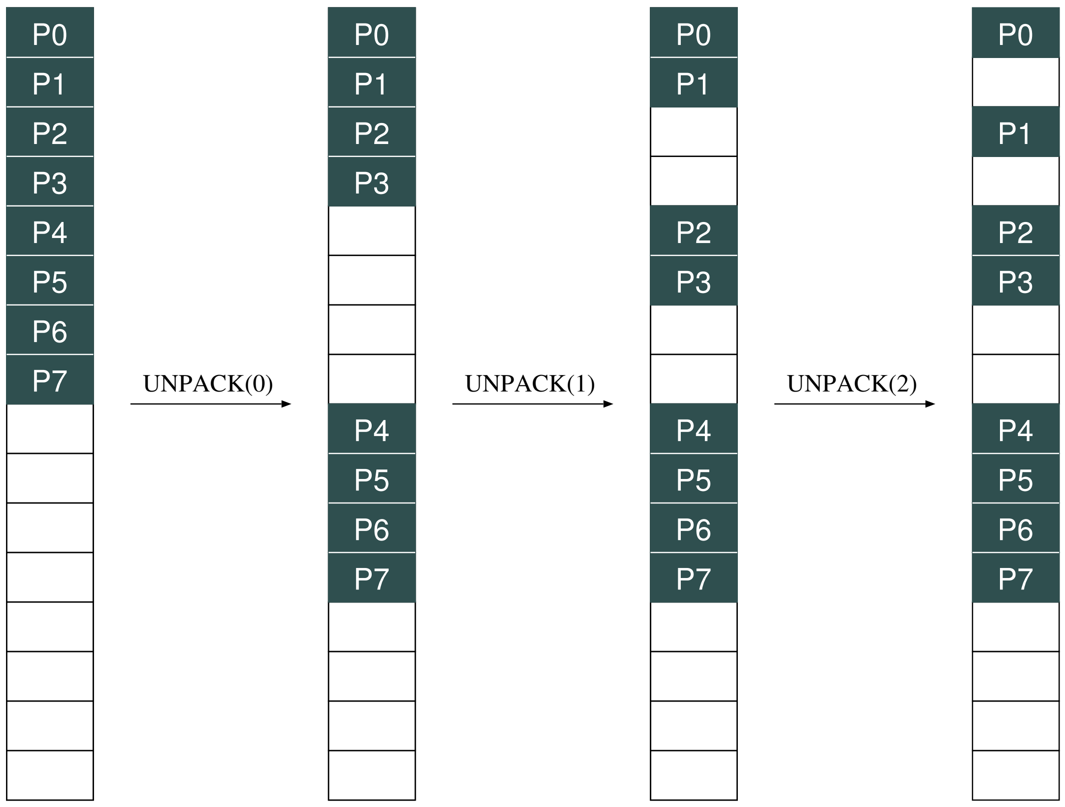

Figure 5.1: snapshots of the BT memory layout during an UNPACK(0) operation. The host D-BSP machine has 8 virtual processors. Grey boxes indicate processor contexts, and white boxes indicate empty buffers.

the invariant that the topmost i-cluster is followed by a free memory area of size $\mu v/2^i$; as a consequence, the execution time is

$$O\left(\sum_{j=i}^{\log v-1}\left(f(\mu v/2^j)+\mu v/2^{j+1}\right)\right).$$

Since $f(x)$ is polynomially bounded and $f(x) = O(x)$, the execution time of PACK(i) and UNPACK(i) can be simply rewritten as

$$O\left(\frac{\mu v}{2^i}\right). \tag{5.7}$$

Another source of inefficiency of the simulation scheme in Section 5.4 is that message delivery is a straightforward adaptation of the mode of operation of the parallel machine: this is unnecessary, and in fact the independent delivery of every single message limits the exploitation of block transfer. In Subsection 5.5.2, we will see how this issue can be dealt with through a suitable sorting procedure. As far as memory is concerned, we must say that this procedure requires the presence of a certain number of *tags* inside each processor context: the analysis will show that such tags do not alter the time to access any memory cell by more than a constant, multiplicative factor, hence their presence will be overlooked in all the steps of the simulation but the one which uses them.

5.5.2 The Simulation Algorithm

The simulation scheme requires, within a constant factor, $\mu v \log\log \mu v$ memory cells to function properly; cells $0, \ldots, \mu v - 1$ are initially occupied, as usual, by the v processors' contexts, while the remaining cells are empty. Since it is not restrictive, we will suppose that the parallel program $\mathcal{P}$, written for the host D-BSPB $(v, \mu, f(x))$ machine, is $\mathcal{G}(v, \mu, f(x))$-smooth. For ease of presentation, we will further suppose that μ is a power of 2. The pseudo-code of the scheme is as follows.

```
1    UNPACK(0)
     **while** true **do**
2        $P \leftarrow$ processor whose context is on top of memory
         $s \leftarrow$ superstep number to be simulated next for $P$
         $C \leftarrow i_s$-cluster containing $P$
3        PACK($i_s$)
4        COMPUTE($|C|$)
5        simulate the message exchange for $C$
6        **if** $P$ has finished its program **then exit**
7        **if** $i_{s+1} < i_s$ **then**
             Let $\hat{C}$ be the $i_{s+1}$-cluster containing $C$, and let $\hat{C}_0, \ldots, \hat{C}_{2^{i_s - i_{s+1}} - 1}$
             be its component $i_s$-clusters, with $C = \hat{C}_j$ for some index $j$
7.1          **if** $j > 0$ **then** swap the contexts of $C$ with those of $\hat{C}_0$
7.2          **if** $j < 2^{i_s - i_{s+1}} - 1$ **then** swap the contexts of $\hat{C}_0$ with those of $\hat{C}_{j+1}$
8        UNPACK($i_s$)
```

Observe that the overall structure of the simulation is the same as the one presented in Sections 4.4 and 5.4: in particular, the above scheme simulates one cluster per round, and the choice of such cluster is based on the same "proximity" criterion we have already discussed. As a consequence, it is unnecessary to include a new proof of correctness. Furthermore, the code maintains the invariant that at the beginning of each round, the overall memory layout is the same as the one resulting from a call to UNPACK(0).

As indicated by Steps 4 and 5 of the pseudo-code, the simulation of Superstep s for cluster C is performed in two phases: first, local computations are executed in a recursive fashion, and then the communications required by the superstep are simulated. By Step 3 of the algorithm, the simulation of Superstep s for cluster C begins with all contexts of C being packed at the top of memory. In order to exploit both temporal and spatial locality, processors' contexts are iteratively brought to the top of memory in chunks of size $c(v/2^{i_s}, \mu)$, and the prescribed local computation is then performed for each chunk recursively. Memory movements can be performed by exploiting block transfer because of the buffer of size $\mu v/2^{i_s}$ that is available immediately after the cluster. Eventually, each processor context reaches the topmost μ memory cells where computation finally takes place. This step of the simulation is described by the recursive subroutine COMPUTE(n), whose pseudo-code is given in Figure 5.2; the subroutine is initially invoked with parameter $v/2^{i_s}$. It is essential to note that the buffer is big enough to accommodate for the recursive calls. Let $f^{(k)}(x)$ be the *iterated function* which is obtained by applying $f(x)$ k times, and let $f^*(x) = \min\{k \geq 1 : f^{(k)}(x) \leq \mu\}$. Finally, let $c(x) = c(x, \mu)$. Since $c(n) \leq n/2$, the overall buffer space required by COMPUTE(n) and all its recursive calls is smaller than

$$\mu \sum_{k=1}^{c^*(v/2^i)} n/2^k \leq \mu n.$$

In particular, COMPUTE($v/2^{i_s}$) needs a buffer of size at most $\mu v/2^{i_s}$, which is certainly available after Step 3.

Let us now calculate the execution time of COMPUTE($v/2^{i_s}$). As usual, local computations for a processor are always performed while the corresponding processor context is stored in the topmost μ memory cells, hence the time for local

```
        COMPUTE($n$)
4.1     **if** $n = 1$ **then** {we were told to operate on a single context}
4.2         Simulate local computation for the context in block 0
        **else**
4.3         $c \leftarrow c(n, \mu)$
4.4         $t \leftarrow n/c$ {number of chunks}
4.5         Shift blocks $c, \ldots, n-1$ to blocks $2c, \ldots, n+c-1$
            {creates buffer space for recursive calls}
4.6         COMPUTE($c$)
4.7         **for** $j \leftarrow 2$ **to** $t$ **do**
4.8             Swap blocks $0, \ldots, c-1$ with blocks $jc, \ldots, (j+1)c-1$
4.9             COMPUTE($c$)
4.10            Swap blocks $jc, \ldots, (j+1)c-1$ with blocks $0, \ldots, c-1$
4.11        Shift blocks $2c, \ldots, n+c-1$ to blocks $c, \ldots, n-1$
```

Figure 5.2: the COMPUTE subroutine.

computations is simply the sum of the original computation times for the guest processors belonging to cluster C. In addition to this cost, COMPUTE(n) presents a significant overhead $T_{\mathrm{MEM}}(n)$ caused by memory copy operations. Since each shift or swap operation requires a constant number of block transfers, it is easy to see that

$$T_{\mathrm{MEM}}(n) = \begin{cases} \frac{n}{c(n)} T_{\mathrm{MEM}}(c(n)) + O\left(\mu n + \sum_{j=1}^{n/c(n)} f\left(j\mu c(n)\right)\right) & \text{for } n > 1, \\ O\left(1\right) & \text{for } n = 1. \end{cases}$$

(Note that $n/c(n)$ is an integer number since both n and $c(n)$ are powers of 2.) The solution of the recurrence equation depends on $f(x)$; we now give an upper bound on $T_{\mathrm{MEM}}(n)$ that holds for $f(x) = O\left(x\right)$, which encompasses the class of functions we are considering throughout this section. First of all, the summation that appears in the recurrence equation is $O\left((n/c(n))f(\mu n)\right)$. By Inequality (5.6), $c(n)$ is not smaller than $\min\{f(\mu n)/\mu, n/2\}/2$, hence the summation is $O\left(\mu n\right)$ and the recurrence equation can be rewritten as

$$T_{\mathrm{MEM}}(n) = \begin{cases} \frac{n}{c(n)} T_{\mathrm{MEM}}(c(n)) + O(\mu n) & \text{for } n > 1, \\ O(1) & \text{for } n = 1. \end{cases} \tag{5.8}$$

The bound we give on $T_{\mathrm{MEM}}(n)$ is based on the reckoning of the work that is performed at each level of the recursion. By direct inspection of Equation (5.8), it is straightforward to see that such a work is $O(\mu n)$ at the base level, and $(n/c(n))O(\mu c(n)) = O(\mu n)$ at the first level. In general, the work at the j-th level of recursion can be expressed through the iterated function $c^{(k)}(x)$ as

$$O\left(\mu c^{(j)}(n)\right) \prod_{k=1}^{j} \frac{c^{(k-1)}(n)}{c^{(k)}(n)},$$

which is still $O(\mu n)$ for $1 \leq j \leq c^*(n)$. By summing up the contributions of the $c^*(n) + 1$ levels of recursion, we conclude that

$$T_{\mathrm{MEM}}(n) = O(\mu n c^*(n)).$$

As particular instances, $T_{\mathrm{MEM}}(n)$ is $O(\mu n \log^* n)$ if $f(x) = \log x$, and $O(\mu n \log\log n)$ if $f(x) = x^\alpha$. For the purpose of our simulation, it is enough to say that if $f(x) = O(x^\alpha)$, then $c^*(n) = O(\log\log n)$ and consequently

$$T_{\mathrm{MEM}}(v/2^{i_s}) = O\left(\frac{\mu v}{2^{i_s}} \log\log \frac{v}{2^{i_s}}\right).$$

The second part of the simulation of cluster C is the delivery of messages to destination processors (Step 5). To perform this step, the contexts of C are divided into $\Theta(\mu|C|)$ constant-sized elements, which are attached a tag; the elements are then sorted according to the tags. If the tags are suitably chosen, the elements are delivered in such a way that the structure of the contexts is not disrupted, and D-BSPB messages reach the input buffers of the appropriate processors. Although the idea behind this simulation step is quite simple, there are two technical issues that must be dealt with.

1. The sorting algorithm must use block transfer effectively. We select the algorithm of [ACS87] for this purpose; anyway, as we will see, this algorithm requires space $O(n \log\log n)$ to sort n elements, hence extra space must be provided in some way.

2. The sorting algorithm may alter the position and size of some contexts, and tags must be removed; it is therefore necessary to introduce a final "clean up" step.

Let us start by describing the tagging procedure. As anticipated, the context of each processor $P_i \in C$ is divided into constant-sized elements; each element is attached a tag, which is a triple (a, b, c) of integer numbers. Let $\kappa < \mu$ be the size of an element, and let λ be the size (to be determined) of a tag. For ease of presentation, we also suppose that a processor P_i, $0 \leq i < v$, exchanges messages through a unified input/output queue which resides at the end of P_i's context. Tagging for P_i is performed in accordance with the following rules.

- The part of P_i's context not belonging to the queue is divided into elements of exactly κ memory cells each; the last element may require padding. Element l of this part receives $(i, -1, l)$ as tag.

- In the I/O queue, each outgoing message is divided into elements of exactly κ memory cells each; the last element of each message may require padding. Element l of the queue is tagged as (j, i, l), where P_j is the destination processor of the message that originated the element. Memory cells that do not contain messages (i.e. the empty part of the queue) are all assigned the dummy tag $(v, v, 0)$.

Note that the tagging procedure increases the amount of memory required to store a context because tags are added and some elements may require padding. To be precise, since the size of a tag is λ and at most $h + 1$ elements require padding, the maximum size of a cluster after tagging is $(\lambda+\kappa)\lceil\mu/\kappa\rceil+(h+1)\kappa = O(\lambda\mu)$; to make this bound meaningful, it is necessary to know λ. The first two components of a tag assume values not greater than v, and the third component is at most $\lceil\mu/\kappa\rceil$: as a consequence, $O(\log v + \log \mu)$ bits are sufficient to store the tag. We can reasonably assume that the host machine is powerful enough to represent μ or v in a single memory word, so the size of λ is bounded by a constant and the size of a context after tagging is $O(\mu)$. Although the contexts are not asymptotically bigger after tagging, they certainly require more than μ cells each to be stored. However, this is not a problem because $f(x)$ is polynomially bounded so we can manage $\Theta(\mu)$ extra memory cells after each of the v contexts while increasing the execution time by a constant multiplicative factor.

To conclude the analysis of the tagging procedure, we must determine its running time. For any i, the tags belonging to the first part of the context of P_i depend only on i, so they do not change during the simulation and they can be statically assigned before the simulation begins. Instead, the tags for the I/O queue depend on the particular communication pattern that is being performed, hence they must be necessarily created on-line: we choose to create them just after the simulation of the local computations for P_i, when the context of P_i is still in the fastest cells at the top of memory. This operation requires a constant amount of work on each of the topmost $\Theta(\mu)$ memory cells. As a whole, the tagging procedure for cluster C is not more costly than touching the $|C|$ contexts belonging to the cluster; by Proposition 5.3.1, such a touching problem can be solved in time

$$O(\mu|C|\log\log(\mu|C|)) = O\left(\frac{\mu v}{2^{i_s}}\log\log\frac{\mu v}{2^{i_s}}\right) \tag{5.9}$$

on a $O(x^\alpha)$-BT machine.

Let us now move to the sorting stage. We adopt the *Approx-Median-Sort* algorithm proposed in [ACS87]; this algorithm is capable of sorting m constant-sized items in time $O(m\log m)$ if $f(x) = O(x^\alpha)$, $0 \le \alpha < 1$. Unfortunately, the algorithm requires $O(m\log\log m)$ space. In our case, the number of elements to sort is $\Theta(\mu v/2^{i_s})$ so the required time is

$$O\left(\frac{\mu v}{2^{i_s}}\log\frac{\mu v}{2^{i_s}}\right), \tag{5.10}$$

and the required memory space $L = L(i_s)$ is

$$O\left(\frac{\mu v}{2^{i_s}}\log\log\frac{\mu v}{2^{i_s}}\right). \tag{5.11}$$

Our buffer policy ensures that when we start simulating the message exchange, the i_s-cluster is followed by an empty space of size $\mu v/2^{i_s}$: clearly this is not enough, because it is asymptotically smaller than (5.11). To obtain more free space, we are forced to involve a cluster bigger than C in the sorting stage. Recall that the buffer creation policy ensures that for every $0 \le i \le i_s$, if we PACK the topmost

i-cluster, we create an adjacent free memory region having the same size of the cluster: intuitively, if i is small enough then $\mu v/2^i \geq L$. More formally, let $i_k < i_s$ be the biggest integer such that $\mu v/2^{i_k} \geq L(i_s)$, or 0 if $\mu v < L(i_s)$. Then, we can free a sufficient amount of space for sorting through the following steps.

```
A.1   UNPACK($i_s$)
A.2   PACK($i_k$)
A.3   Shift blocks $v/2^{i_s}, \ldots, v/2^{i_k} - 1$ to the memory region
      that starts with block $v/2^{i_s} + \lceil L(i_s)/\mu \rceil$
```

After the execution of the above fragment of pseudo-code, memory cells $\mu v/2^{i_s}, \ldots,$ $\mu v/2^{i_s} + L - 1$ are free and available for the Approx-Median-Sort algorithm. By (5.7), the time for the execution of Steps A.1 and A.2 is

$$O\left(\mu v/2^{i_k}\right) = O\left(L\right) = O\left(\frac{\mu v}{2^{i_s}} \log\log \frac{\mu v}{2^{i_s}}\right).$$

Step A.3 move a block of at most L memory cells into an empty memory region; the index of the slowest cell belonging to this region is no greater than $2L$. Since $f(x) = O\left(x\right)$, the time required to perform the prescribed memory movement is therefore

$$O\left(f(L) + L\right) = O\left(L\right) = O\left(\frac{\mu v}{2^{i_s}} \log\log \frac{\mu v}{2^{i_s}}\right),$$

that is, not asymptotically higher than the time (5.10) required by the Approx-Median-Sort algorithm.

Finally, we must examine the clean up stage, which must clearly undo all the undesired effects of the previous stages. Tags, padding data and memory buffers can be removed by performing the operations that created them in reverse order; besides, there is a side-effect of sorting that must be carefully considered. Remember that, during the tagging stage, the portions of input/output buffers that contained no messages were all marked with the same tag $(v, v, 0)$, so Approx-Median-Sort places them after all the contexts of C. As a consequence, after the sorting phase the input/output buffer of a processor has a size that depends on the aggregate size of the messages it received. Since each processor may receive a different amount of data, each context of C may have a different size, thus occupying a position that is different from the original one by a nonconstant amount; this is a problem, since the simulation algorithm relies on the knowledge of the position of the contexts. Before going on with the simulation, it is therefore necessary to realign the contexts so that the context of the j-th processor of C ends up again in memory block j, for $0 \leq j < |C|$. Since the contexts C are still in sorted order (i.e. context l comes first, then context $l+1$ and so on, for a suitable integer l), this operation can be performed by the recursive subroutine that follows.

```
      ALIGN($n$)
B.1   **if** $n = 1$ **then exit**
B.2   Locate the $(n/2)$-th topmost context
B.3   Shift contexts $n/2, \ldots, n-1$ to the memory region
      that starts with block $n$
B.4   ALIGN($n/2$)
B.5   Swap blocks $0, \ldots, n/2-1$ with blocks $n, \ldots, 3n/2-1$
B.6   ALIGN($n/2$)
B.7   Shift blocks $0, \ldots, n/2-1$ to blocks $n/2, \ldots, n-1$
B.8   Swap blocks $n, \ldots, 3n/2-1$ with blocks $0, \ldots, n/2-1$
```

The subroutine is initially invoked with $n = |C| = v/2^{i_s}$. The location of a context can be accomplished through binary search over the tags. Each step of the binary search copies a block of $2(\kappa+\lambda) = O(1)$ consecutive memory cells on top of memory and then scans them until it finds a tag (the size of the block is chosen so that it certainly contains at least one tag); the tag is associated with a single processor context, and with a specific position inside that context. If the tag identifies the beginning of context $v/2^{i+1}$ then the search is over, otherwise the search continues on a halved search area. The binary search requires $O(\log(\mu n))$ steps; each such step reads a constant number of consecutive memory cells and then perform a constant amount of work on them, thus taking time $O(f(\mu n))$. Finally, context realignment (Step B.3) and the swaps of the two recursive subproblems (Steps B.5 and B.7) are all implemented with pipelined memory copy operations and require time $O(f(\mu n) + \mu n)$ as a whole. By putting all these observations together, the time $T_{\mathrm{AL}}(n)$ for alignment can be distilled into the following recurrence equation:

$$T_{\mathrm{AL}}(n) = \begin{cases} 2T_{\mathrm{AL}}(n/2) + O\left(f(\mu n)\log(\mu n) + \mu n\right) & \text{for } n > 1, \\ O(1) & \text{for } n = 1. \end{cases}$$

Since $f(x) = O(x^\alpha)$, $\alpha < 1$, the recurrence equation evaluates to $O(\mu n \log(\mu n))$, therefore the time taken by the clean up stage for an i_s-cluster is

$$O\left(\frac{\mu v}{2^{i_s}} \log \frac{\mu v}{2^{i_s}}\right). \tag{5.12}$$

Having completed the analysis of all the steps, we can now give an upper bound on the simulation time of Superstep s for cluster C. The time spent for PACK(i_s) and UNPACK(i_s) operations is given by Equation (5.7). The time for local computations is simply the sum τ of computation times for the processors in C. The time for message delivery is the sum of the execution times for the tagging, sorting and clean up steps, which are given by Equations (5.9), (5.10) and (5.12) respectively. As a whole, the time required to simulate i_s-cluster C is therefore

$$O\left(\tau + \frac{\mu v}{2^{i_s}} \log \frac{\mu v}{2^{i_s}}\right)$$

under the hypothesis $f(x) = O(x^\alpha)$, $0 \leq \alpha < 1$. It is important to note that this bound *does not* depend on $f(x)$ and h any more. Note also that the simulation of communications is dominated by the sorting step.

If $i_{s+1} < i_s$, the simulation algorithm incurs an additional cost due to the need of moving a new cluster to the top of memory, as prescribed by Steps 7 through 7.2 of the pseudo-code. Thanks to the availability of buffer space, the swaps of Steps 7.1 and 7.2 can be performed with three pipelined memory copy operations each, thus taking time $O(f(j\mu|C|) + \mu|C|)$. By summing up this cost over all the values of j, we conclude that the overall time for cluster swaps before the simulation of an i_{s+1}-cluster is

$$O\left(\sum_{j=1}^{2^{i_s - i_{s+1}}} \left(f(j\mu|C|) + \mu|C|\right)\right) = O\left(2^{i_s - i_{s+1}} f\left(\frac{\mu v}{2^{i_{s+1}}}\right) + \frac{\mu v}{2^{i_{s+1}}}\right). \tag{5.13}$$

Now, by the $\mathcal{G}$-smoothness of the program $\mathcal{P}$ being simulated, i_s and i_{s+1} are consecutive indices in $\mathcal{G}(v, \mu, f(x))$, hence $f(\mu v/2^{i_{s+1}}) = \Theta(f(\mu v/2^{i_s}))$ because of (4.2).

By applying this observation together with the fact that $f(x) = O(x)$, we conclude that the overall cost of cluster swaps is $O\left(\frac{\mu v}{2^{i_{s+1}}}\right)$, hence it is amortized by the cost of the future simulation of the i_{s+1}-cluster. As a consequence, the cost of Step 7 can be completely neglected and the time for simulating $\mathcal{P}$ is simply the sum of the times for the simulation of the clusters that compose the various supersteps of $\mathcal{P}$.

Theorem 5.5.2 *Consider a program $\mathcal{P}$ for D-BSPB $(v, \mu, f(x))$, where each processor performs local computation for $O(\tau)$ time, and there are λ_i i-supersteps for $0 \le i \le \log v$. If $f(x) = O(x^\alpha)$, $0 \le \alpha < 1$, then $\mathcal{P}$ can be simulated on $f(x)$-BT in time*

$$O\left(v\left(\tau + \mu\sum_{i=0}^{\log v} \lambda_i \log(\mu v/2^i)\right)\right).$$

Theorem 5.5.2 shows that the advanced simulation scheme is quite efficient for full programs. In addition to local computation time, which exhibits a slowdown that is merely proportional to the loss of parallelism and is consequently optimal, the scheme incurs a cost per superstep that is $O(\mu v \log(\mu v))$ in the worst case[1]. For the sake of comparison, Proposition 5.3.1 implies that for relevant access functions $f(x)$, any straightforward approach simulating one entire superstep after the other would require time $\omega(\mu v)$ per superstep just for touching the v processor contexts. Note that the scheme does no longer offer a "Brent-like" slowdown in the sense of Corollaries 4.4.8 and 5.4.3; this fact will be discussed in Section 5.6.

5.5.3 Partial Loss of Parallelism

Let us now see how the advanced simulation scheme can be employed if the simulating (host) machine has $v' > 1$ processors, i.e. the loss of parallelism is only partial. The simulation strategy is still the one that is described in the proof of Theorem 4.4.3. Recall that for every $0 \le j < v'$, the processors in cluster $C_j^{(\log v')}$ of the guest machine are assigned to host processor P_j. For $i < \log v'$, each i-superstep of $\mathcal{P}$ is simulated by an i-superstep on the host. Such a superstep can be simulated in time

$$O\left((v/v')(\tau' + hf(\mu v/2^i) + \mu)\right),$$

[1]Actually, it may be asymptotically less than this if the superstep label i is greater than zero.

where the term $(v/v')(hf(\mu v/2^i) + \mu)$ accounts for the execution of an (hv/v')-relation and dominates the time $O\left((v/v')(f(\mu v/v') + \mu)\right)$ required by memory movements during the simulation of the v/v' local computations at each host processor. Instead, for $i \geq \log v'$, each i-superstep is simulated sequentially, using the strategy of Subsection 5.5.2. The slowdown incurred in the simulation of these supersteps is

$$O\left((v/v')\left(\tau' + \mu \log\left(\frac{\mu v/v'}{2^i}\right)\right)\right),$$

and is obtained by applying Theorem 5.5.2 while setting the number of processors of the guest machine to v/v'.

To sum things up, the simulation time exhibits a flex point for $i = \log v'$. For superstep indices smaller than $\log v'$, communications are simulated by resorting to the network of the host machine, thus exhibiting a (v/v')-fold slowdown. On the other hand, if $i \geq \log v'$, the advanced simulation scheme is used with all the consequences that follow. Therefore, this simulation strategy leads to the following result.

Proposition 5.5.3 *Consider a program $\mathcal{P}$ for D-BSPB $(v, \mu, f(x))$, where each processor performs local computation for $O(\tau)$ time, and there are λ_i i-supersteps for $0 \leq i \leq \log v$. If $f(x) = O(x^\alpha)$, $0 \leq \alpha < 1$, then for any $1 \leq v' \leq v$, $\mathcal{P}$ can be simulated on D-BSPB $(v', \mu v/v', f(x))$ in time*

$$O\left(\frac{v}{v'}\left(\tau + \mu \sum_{i=0}^{\log v'-1} \lambda_i f(\mu v/2^i) + \mu \sum_{i=\log v'}^{\log v} \lambda_i \log\left(\frac{\mu v/v'}{2^i}\right)\right)\right).$$

5.6 Discussion and Applications

Since we presented two simulation schemes, it is natural to wonder if what we name the "advanced" scheme is indeed more efficient than the plain one, and under which conditions. To address this question, in this section we perform a comparison of the two schemes. We concentrate on the simulation of communications since both schemes simulate local computations optimally. Since the advanced scheme is tailored to $f(x)$-BT, the comparison is limited to the case in which there is a complete loss of parallelism.

The first observation is against the advanced scheme: if $f(x) = o(\log x)$, in fact, communications are simulated more efficiently by the plain scheme. This conclusion is obtained by comparing execution times as expressed by Theorems 5.4.1 and 5.5.2. The limiting factor of the advanced scheme relies in the use of sorting: in fact, sorting $\Theta(\mu v)$ items takes time $\Theta(\mu v \log(\mu v))$ even on the flat-memory RAM model. Although the use of sorting may look unjustified, note that [ACS87, Corollary 5.9] shows that the expected time for achieving a random permutation on n elements is $\Omega(n \log n)$ on a $f(x)$-BT machine with $f(x) = \Omega(\log x)$. With high probability, for a wide class of cost functions there is nothing to be gained by looking at message delivery as a permutation. These observations provide evidence that if the memory cost function $f(x)$ grows very slowly, then it is profitable to send messages directly to their destinations instead of moving them multiple times according to elaborate, recursive algorithms.

If $f(x) = \Omega(\log x)$, let $n \stackrel{\Delta}{=} \mu v$ be the aggregate size of the memory available in the guest machine for the execution of program $\mathcal{P}$. In most situations n can also be regarded as the size of the problem to be solved by $\mathcal{P}$, so it is interesting to see how the two simulation schemes behave "asymptotically", i.e. as n grows. We are interested in two main scenarios.

1. **Limited parallelism.** The number v of processors remains constant as n grows; in other words, we can say that $v = O(1)$ and $\mu = \Theta(n)$. This scenario is "realistic", in the sense that any real-world parallel computer has a fixed number of processing units. In this scenario, block transfer potentially offers a significant advantage.

2. **Full Parallelism.** The number of virtual processors is a constant fraction of n: as a consequence, $v = \Theta(n)$ and $\mu = O(1)$. This scenario reflects our view of the guest machine as a virtual model that conveniently guides the choices of the algorithm designer: the exploitation of the memory hierarchy is then left to the automatic translation of parallelism into locality of reference. Note that, in this scenario, D-BSP$(v, 1, f(x))$ and D-BSPB $(v, 1, f(x))$ are equivalent within constant factors, since block transfer in the latter model cannot offer but a constant speed-up.

In the first scenario, the time spent for message delivery during an i-superstep is

- $O(hf(\mu) + \mu) = O(hf(n) + n)$ for the plain simulation scheme, and
- $O(\mu \log \mu) = O(n \log n)$ for the advanced simulation scheme.

Recall that h is the maximum number of messages exchanged by a processor. Indeed, h decides which scheme is more efficient: if $h = o(n \log n / f(n))$, then the plain scheme is faster, otherwise the advanced scheme is faster. Note that the plain scheme may be sometimes preferable even if $\mathcal{P}$ is full; a notable evidence of this fact is the case of full programs where $h = O(1)$ in every superstep. For example, the algorithms of Section 4.5 naturally fall in this class if adapted to the limited parallelism scenario.

In the second scenario (full parallelism), the time spent for message delivery during an i-superstep sums up to

- $O(vf(v/2^i)) = O(nf(n/2^i))$ for the plain simulation scheme, and
- $O(v \log(v/2^i)) = O(n \log(n/2^i))$ for the advanced simulation scheme.

(Remember that $h \leq \mu$, and that $f(x)$ is polynomially bounded.) In this case, the advanced scheme is always convenient: as a matter of fact, the faster $f(x)$ grows, the bigger the improvement. Moreover, the advanced scheme offers the potential for sublinear slowdown if applied to communication-bound problems. For example, consider the problem of multiplying two $\sqrt{n}$ by $\sqrt{n}$ dense matrices; the guest and host machines are the D-BSP$(n, 1, x^\alpha)$ and the x^α-BT, $0 < \alpha < 1$. We resort to the recursive algorithm described in Section 4.5; the running time on D-BSP is given by Proposition 4.5.1 and is

$$T_{\mathrm{MM}}(n) = \begin{cases} O(n^\alpha) & \text{for } 1/2 < \alpha < 1, \\ O(\sqrt{n} \log n) & \text{for } \alpha = 1/2, \\ O(\sqrt{n}) & \text{for } 0 < \alpha < 1/2. \end{cases}$$

With the help of recurrence equation (4.3), it is easy to prove that this algorithm uses superstep label $2i$, $0 \leq i \leq (\log n)/2$, a number of times proportional to 2^i: by

Theorem 5.5.2, the running time of the algorithm on the host machine is therefore

$$O\left(n^{3/2} + n \sum_{i=0}^{(\log n)/2} 2^i \log(n/2^{2i})\right) = O\left(n^{3/2}\right). \tag{5.14}$$

Then, for the matrix multiplication problem, the advanced simulation scheme yields an optimal sequential algorithm. Note that a trivial step-by-step D-BSP simulation would have required at least time $\Omega\left(n^{3/2} \log\log n\right)$, hence the exploitation of network locality is crucial in this case. Besides, the slowdown incurred by the advanced scheme with respect to the parallel algorithm is

$$\begin{cases} O\left(n^{3/2-\alpha}\right) & \text{for } 1/2 < \alpha < 1, \\ O\left(n/\log n\right) & \text{for } \alpha = 1/2, \\ O\left(n\right) & \text{for } 0 < \alpha < 1/2. \end{cases}$$

As a consequence, for $1/2 \leq \alpha < 1$, our simulation also exhibits a slowdown that is less than proportional to the loss of parallelism.

The main reason why matrix multiplication exhibits good results is that, on the host machine, the overall amount of computation is asymptotically higher than any other cost. In other words, the algorithm performs enough computation to hide the "inefficiency" of the simulation, which is caused by memory movements and is quantitatively determined by the summation

$$\mu v \sum_{i=0}^{\log v} \lambda_i \log(\mu v/2^i) \tag{5.15}$$

in Theorem 5.5.2. Since the amount of computation is optimal in this parallel algorithm, and the advanced scheme simulates local computation optimally, the algorithm produced by the simulation is also optimal. A sublinear slowdown is attained for matrix multiplication because, on the guest machine, the computation and communication costs are unbalanced in the opposite direction: in other words, for $\alpha \geq 1/2$ the cost function x^α is such that, on the guest machine, communication time dominates the time τ spent for computation. In general, we can say that a sublinear slowdown is observed in the simulation of a D-BSPB $(v, \mu, f(x))$ program on $f(x)$-BT if the following two conditions hold:

1. $\sum_{i=0}^{\log v} \lambda_i f(\mu v/2^i) = \omega\left(\tau\right)$;

2. $\mu \sum_{i=0}^{\log v} \lambda_i \log(\mu v/2^i) = O(\tau)$.

These observations on optimality and slowdown show that the key to the performance of the simulation is summation (5.15), which includes all the costs of the simulation but local computation. Anyway, such observations do not bring practical clues to the properties that must be exhibited by the parallel algorithm *and* the parallel model to ensure an optimal, sequential algorithm. In particular, note that it is not always true that if a parallel algorithm for D-BSPB $(v, \mu, f(x))$ is *work-optimal*, then it can be simulated in optimal time through the advanced scheme.

Definition 5.6.1 *The* work *performed by a v-processor algorithm, $v \geq 1$, is the product of its running time and v. A parallel algorithm is* work-optimal *with respect to a sequential model M_{seq} if its work is within a constant factor from the one of an optimal algorithm on M_{seq} for the same problem.*

This is a weaker definition of work-optimality than the one that is commonly adopted (see e.g. [JáJ92]), which assumes a RAM [AHU74] as the sequential model: in the light of Proposition 5.3.1, we feel that it is unfair to take as golden standard the running time on a model where the cost of accesses to memory can be completely neglected. By Definition 5.6.1 and Corollary 5.4.2, we can say that if a full (not necessarily fine-grained) parallel program is work-optimal with respect to $f(x)$-BT, then the plain scheme turns it into a sequential $f(x)$-BT program that is also optimal. As we have already explained, the same property may not hold for the advanced scheme, even under our weak notion of work-optimality: for instance, this is the case if the optimal parallel algorithm is computation-bound, while the sequential running time obtained through simulation is dominated by the cost (5.15) of memory movements.

At this point, it is natural to wonder which parallel model is best "coupled" with (5.15): in other words, which set of parameters for D-BSPB yields the best prediction for the running time of the simulation? Which set guides algorithmic choices in such a way that the simulation gives the best running time possible? Unlike the HMM scenario of Chapter 4, the straightforward choice of having the same cost function for the guest and the host machine is not optimal. Consider, for example, the problem of solving an n-DFT instance on x^α-BT, for $0 < \alpha < 1$; the guest machine that is used for algorithm development is the D-BSP$(n, 1, x^\alpha)$. As for many problems in this section, we reuse the algorithms of Section 4.5. Remember that Section 4.5 presents two n-DFT algorithms: the first one is a standard execution of the n-input FFT dag, while the second is based on a recursive decomposition of the same dag into two layers of $\sqrt{n}$ independent $\sqrt{n}$-input subdags, which are assigned to distinct D-BSP clusters. The two layers are separated by a transpose permutation, that is, by a 0-superstep. The running time of the first algorithm on D-BSP$(n, 1, x^\alpha)$ is given by Proposition 4.5.2 and is $O(n^\alpha)$. For the second algorithm, the proof of Proposition 4.5.2 shows that it requires 2^i supersteps with label $(1 - 1/2^i)\log n$, $0 \le i < \log\log n$, hence the running time is

$$O\left(\sum_{i=0}^{\log\log n - 1} 2^i \left(n^{1/2^i}\right)^\alpha\right) = O\left(n^\alpha\right).$$

Clearly, the two algorithm are equally efficient and optimal on D-BSP$(n, 1, x^\alpha)$. However, when the advanced scheme is adopted, the simulation times of these two algorithms on x^α-BT are

$$O\left(n\log n + n\sum_{i=0}^{\log n - 1} \log(n/2^i)\right) = O\left(n\log^2 n\right)$$

and

$$O\left(n\log n + n\sum_{i=0}^{\log\log n - 1} 2^i \log(n^{1/2^i})\right) = O\left(n\log n\log\log n\right),$$

respectively: in this case, the second algorithm is more efficient than the first one. This implies that the choice of having the same cost function for the guest and host machine is not effective, in the sense that D-BSP$(n, 1, x^\alpha)$ does not reward the use of the second algorithm over the first.

To find the most effective cost function for the guest machine, we observe that (5.15) resembles the overall cost of communication for a D-BSP$(v, \mu, \log x)$; actually, if the slowdown v due to the loss of parallelism is neglected, it is that very cost for full algorithms. As a consequence, D-BSPB $(v, \mu, \log x)$ seems the guest model we are looking for. For fine-grained programs, this intuition is substantiated by the following proposition.

Proposition 5.6.2 *Any T-time program $\mathcal{P}$ for D-BSP$(v, 1, \log x)$ can be simulated in time $O(vT)$ on $f(x)$-BT with $f(x) = O(x^\alpha)$, $0 \le \alpha < 1$.*

Proof Our aim is to use the advanced scheme for the simulation. Such scheme needs $\mathcal{P}$ to be $\mathcal{G}(v, 1, f(x))$-smooth: in the general case, this hypothesis can be satisfied only by a transformation that increases the running time of $\mathcal{P}$ by more than a constant factor. To solve this issue, we slightly alter both the program $\mathcal{P}$ and the simulation scheme. As far as the program is concerned, $\mathcal{P}$ is augmented by inserting the minimum number of *dummy supersteps* to enforce the property that any two consecutive supersteps j and $j+1$, with $i_{j+1} < i_j$, satisfy the inequality

$$f(v/2^{i_{j+1}}) \le v/2^{i_j}. \tag{5.16}$$

Consider now two "original" supersteps s and $s+1$ of $\mathcal{P}$ with $i_{s+1} < i_s$ which do not satisfy Inequality (5.16). The number k of dummy supersteps to be inserted is such that $f^{(k)}(v/2^{i_{s+1}}) \le v/2^{i_s}$, whence $k \le f^*(v/2^{i_{s+1}}) = O(\log\log(v/2^{i_{s+1}}))$.

With regard to the simulation, the advanced scheme is modified to treat dummy and normal supersteps differently: to be precise, when the simulation algorithm realizes that the topmost i_j-cluster needs to execute a dummy superstep, it immediately passes to Superstep $j+1$. As a consequence, the only cost incurred in simulating a dummy i_j-superstep is due to the need of moving a certain number of i_j-clusters to the top of memory (Step 7); this cost is given by Equation (5.13) and is

$$O\left(2^{i_j - i_{j+1}} f(v/2^{i_{j+1}}) + v/2^{i_{j+1}}\right),$$

where $i_{j+1} < i_j$ is the index of the superstep which triggers the cluster movements. Since i_j and i_{j+1} satisfy Inequality (5.16), the cost for the simulation of the dummy i_j-superstep is $O(v/2^{i_{j+1}})$. Based on this result, we can say that for every i_{s+1}-cluster, the extra cost for dealing with the $O(\log\log(v/2^{i_{s+1}}))$ dummy supersteps which are placed between non-dummy superstep indices i_s and i_{s+1} is

$$O\left(\frac{v}{2^{i_{s+1}}} \log\log(v/2^{i_{s+1}})\right),$$

which is clearly amortized by the cost of the simulation of the i_{s+1}-superstep.

Having proved that the program transformation applied to $\mathcal{P}$ does not asymptotically increase the simulation time, we can now see how the transformation helps us overcome the fact that $\mathcal{P}$ is not $\mathcal{G}(v, 1, f(x))$-smooth. The simulation scheme of Subsection 5.5.2 relies on the smoothness of $\mathcal{P}$ in the upper bound on the cost of cluster movements, as given by Equation (5.13). The transformation ensures that $f(v/2^{i_s}) = O(v/2^{i_{s+1}})$ for any two consecutive indices i_s, i_{s+1} in $\mathcal{P}$, $i_s < i_{s+1}$; it is straightforward to see that this property can be used to confirm the conclusions we reached in Subsection 5.5.2.

To conclude the proof, we note that the simulation slows down local computation by a factor $vf(\mu)/\log\mu$, where μ is the size of the local memory that is available to each D-BSP processor. Since $\mu = O(1)$, the slowdown is proportional to v. $\square$

Since the running times of a program for D-BSP$(v, 1, \log x)$ and of the corresponding simulation on $f(x)$-BT are related according to a fixed multiplicative factor v, the effectiveness of D-BSP$(v, 1, \log x)$ is ensured: if algorithm $\mathcal{A}_1$ is faster than algorithm $\mathcal{A}_2$ on D-BSP, the inequality is certainly preserved if the two algorithms are simulated on $f(x)$-BT. Furthermore, for the purposes of this simulation no cost function suggests faster sequential algorithms than $\log x$. Let $\mathcal{C}$ be a computational problem, $g(x) \neq \log x$ a polynomially bounded function, and $\mathcal{P}_g$, $\mathcal{P}_{\log}$ two optimal programs for $\mathcal{C}$ on D-BSP$(v, 1, g(x))$ and D-BSP$(v, 1, \log x)$, respectively. Note that $\mathcal{P}_g$ is also a valid program for D-BSP$(v, 1, \log x)$: on this model it is therefore possible to evaluate the running times T_g, $T_{\log}$ of $\mathcal{P}_g$ and $\mathcal{P}_{\log}$, respectively. Now, suppose that $\mathcal{P}_g$ yields, once simulated, a sequential algorithm that is asymptotically faster than the one given by $\mathcal{P}_{\log}$; by Proposition 5.6.2, the running times of the two simulations are vT_g and $vT_{\log}$, respectively. The first quantity is asymptotically smaller than the second one, so we can immediately conclude that $T_g = o(T_{\log})$: this is a contradiction, since we defined $T_{\log}$ to be the optimal time for $\mathcal{C}$ on D-BSP$(v, 1, \log x)$. The contradiction proves that our claim is true.

By relying on Proposition 5.6.2, we have therefore shown that D-BSP$(v, 1, \log x)$ is the model of choice for the development of sequential algorithms for a whole class of BT machines, namely those in which the cost to access the hierarchical memory is bounded by a polynomial x^α, $0 \leq \alpha < 1$. The argument extends to an arbitrary memory size μ, but only if the D-BSPB $(v, \mu, f(x))$ program is full and $f(x) = \Omega(\log x)$.

Problem	**D-BSP**$(n, 1, \log x)$	$f(x)$**-BT**	**Simulation loss**
n-MM	$O\left(n^{1/2}\right)$	$O\left(n^{3/2}\right)$	1
n-DFT	$O(\log n \log\log n)$	$O(n \log n)$ [ACS87]	$\log\log n$
n-sorting	$O\left(\log^2 n\right)$	$O(n \log n)$ [ACS87]	$\log n$
Broadcast	$O(\log n \log\log n)$	$O(n)$	$\log n \log\log n$

Table 5.1: execution times of some common primitives on the D-BSP$(n, 1, \log x)$ and $f(x)$-BT models, with $f(x) = O(x^\alpha)$, $0 \leq \alpha < 1$. The last column shows the *simulation loss*, that is, the inefficiency incurred when simulating the parallel algorithms on BT, with respect to the best algorithms known for BT.

Note that a single cost function, namely $\log x$, fits the $f(x)$-BT model for all the values of $f(x)$ encompassed by the advanced simulation scheme. This is due to the fact that (5.15) does not depend on $f(x)$, which, in turn, is an effect of block transfer: in fact, the availability of block transfer reduces the average cost of memory accesses enough to make the overhead of the simulation insensitive to $f(x)$. Concerning the parallel model, observe that the logarithmic function describes a network hierarchy in which the cost of communication increases very slowly with the size of the machine. With a tolerable degree of approximation, we can say that the communication subsystem is "fast" and exhibits almost no hierarchy, thus encouraging a massive use of parallelism. Since we use the parallel model that gives the best match with $f(x)$-BT, this is a hint that our simulation scheme operates an effective transformation of parallelism into both temporal and spatial locality.

In the light of Proposition 5.6.2, we now examine the matrix multiplication, DFT, sorting and broadcast problems to determine the running times that are obtained by the simulation on $f(x)$-BT of the best algorithms for the D-BSP$(n, 1, \log x)$ model; the results of the analysis are summarized in Table 5.1.

n-MM By Proposition 4.5.1, the matrix multiplication algorithm we have already described yields a running time of $O\left(n^{1/2}\right)$ on the D-BSP$(n, 1, \log x)$ model: the execution time on $f(x)$-BT is therefore $O\left(n^{3/2}\right)$. A trivial, work-based argument shows that both the parallel and the sequential running times are optimal in the respective computational models.

n-DFT The standard implementation of the n-input FFT dag yields a running time of

$$O\left(\sum_{i=0}^{\log n-1} \log\left(n/2^i\right)\right) = O\left(\log^2 n\right).$$

For the recursive decomposition of the FFT dag into $\sqrt{n}$ independent subdags, the running time is

$$O\left(\sum_{i=0}^{\log\log n-1} 2^i \log(n^{1/2^i})\right) = O\left(\log n \log\log n\right).$$

When the advanced scheme is adopted, the simulation times of these two algorithms on $f(x)$-BT are $O\left(n\log^2 n\right)$ and $O\left(n\log n\log\log n\right)$, respectively. Note that the D-BSP$(n, 1, \log x)$ model is effective, since it favors the second algorithm (which is more efficient on BT) over the first. We further remark that the simulation of the second algorithm, which yields the best known running time on D-BSP$(n, 1, \log x)$, does give a suboptimal algorithm for the $f(x)$-BT model, which can solve the DFT problem in time $\Theta\left(n\log n\right)$ for $f(x) = O\left(x^\alpha\right)$, $\alpha < 1$ [ACS87].

n-sorting To date, the best sorting algorithm for the D-BSP$(n, 1, \log x)$ model is a straightforward implementation of the AKS sorting network [AKS83]: the depth of this network is $O\left(\log n\right)$, hence it can be implemented with $O\left(\log n\right)$ 0-supersteps; the corresponding running time is $O\left(\log^2 n\right)$ on D-BSP$(n, 1, \log x)$, and $O\left(n\log^2 n\right)$ on $f(x)$-BT. Note that the parallel algorithm does not exploit any temporal, spatial or network locality.

Broadcast As explained in Section 2.4, a *broadcast* is a communication operation that delivers a constant-sized item, which is initially stored in a single processor, to every processor of a parallel machine. On D-BSP$(n, 1, \log x)$, we can use a recursive algorithm; when invoked on a problem of size n, the algorithm performs the operations explained below.

1. If $n = 1$, then the call returns without performing any computation.

2. If $n > 1$, then broadcast is recursively solved for the first $(\log n)/2$-cluster of the parallel machine, i.e. for cluster $C_0^{(\log n/2)}$. Then, for $0 < j < \sqrt{n}$, processor P_j sends a copy of the item to processor $P_{j\sqrt{n}}$: when this message exchange is over, every $(\log n/2)$-cluster owns a copy of the item. Finally, broadcast is solved in parallel for submachines $C_1^{(\log n/2)}, \ldots, C_{\sqrt{n}}^{(\log n/2)}$.

As a whole, this algorithm requires two recursive calls on subproblems of size $\sqrt{n}$ and a 0-superstep in which every processor exchanges $O(1)$ messages. The execution time T_{BCAST} of the algorithm is therefore given by the recurrence equation

$$T_{\text{BCAST}}(n) = \begin{cases} 2T_{\text{BCAST}}(n^{1/2}) + O(\log n) & \text{for } n > 1, \\ O(1) & \text{for } n = 1, \end{cases}$$

which evaluates to $O(\log n \log\log n)$.

On the sequential BT model, broadcast can be considered equivalent to the following **Memory Fill** problem: given a value x stored in memory location 0, the value must be copied into the first n memory cells. If $f(x) = O(x^\alpha)$, $0 \le \alpha < 1$, an optimal solution to this problem is organized into $\log n$ steps: during Step j, $0 \le j < \log n$, the interval of memory cells $[0, 2^j - 1]$ is copied onto the interval $[2^j, 2^{j+1} - 1]$ in a single block transfer operation. It is straightforward to prove inductively that, at the beginning of Step j, the value x has been copied into the first 2^j memory cells. The running time of this sequential algorithm is

$$\sum_{j=0}^{\log n - 1} \left(f(2^{j+1} - 1) + 2^j \right) = O(n),$$

which is asymptotically optimal. Instead, the simulation of the parallel algorithm on $f(x)$-BT yields an $O(n \log n \log\log n)$-time program: this is a $\log n \log\log n$ factor away from optimum.

We remark that, although the advanced scheme does not always offer optimal algorithms for the sequential BT model, it cannot be further improved in the general case, since the lower bound proved in [ACS87, Corollary 5.9] on the execution of random permutations on $f(x)$-BT can be employed to design a D-BSP program for which the time of the advanced scheme is the least possible. However, an improved simulation can be obtained when the communication patterns generated by the parallel algorithm are known *a priori* and exhibit certain regularities. As an example, consider again the $O(\log n \log\log n)$-time DFT algorithm designed for D-BSP$(n, 1, \log x)$. By simulating the transpose permutation at each superstep by the rational permutation algorithm in [ACS87], rather than through sorting, the running time of the simulation becomes $O(n \log n)$, which is optimal on $f(x)$-BT for both $f(x) = x^\alpha$ and $f(x) = \log x$. This shows that, in this case, the algorithmic strategy indicated by D-BSP is indeed the optimal one for BT and that non-optimalities are due to the generality of the simulation that must deal with worst case scenarios (e.g., by the use of sorting to cope with random permutations).

Chapter 6

Conclusions

One of the main reasons that makes parallelism difficult to exploit is the lack of a widely accepted model of parallel computation. Countless models are described in the literature, but nearly all of them are focused on some specific aspect of the computation or on specific aspects of the hardware platforms, thus failing to reach a satisfactory balance among usability, portability and effectiveness. This problem has been acknowledged in recent years, and bridging models have been recognized as a viable solution; however, there is still no consensus on the characteristics that must be included in such models. With this thesis we have made a contribution to the growing literature on bridging models, while trying to learn on past mistakes: for this reason, we have focused our attention on what we consider fundamental characteristics of parallel computations, that is, characteristics that are deemed to last even when current technological issues have been solved. This intention has led us to advocate D-BSP [dK96] as a bridging model of parallel computation. The model is characterized by

- a distributed memory architecture;
- a powerful yet easy-to-use communication/synchronization primitive, inherited from the standard BSP;
- a recursive, binary decomposition into independent clusters, which allows the model to capture network locality to an excellent degree without getting involved with the fine details of the network's topology.

In Chapter 2 we have presented some results that demonstrate the higher effectiveness of D-BSP over the popular BSP model according to the quantitative definition of effectiveness introduced in [BPP99]. Although more effective than BSP, D-BSP retains a fair degree of usability, as demonstrated by the simple yet optimal algorithms obtained for the prefix, sorting and routing problems. In the same chapter we have also shown that many processor networks can support the D-BSP abstraction with constant slowdown, provided that the D-BSP parameters are tuned to reflect the bandwidth and latency characteristics of the network under consideration. This fact provides evidence of the virtues of D-BSP as a bridging model, because it shows that D-BSP incorporates the relevant characteristics of interconnection networks. We regard this as a highly desirable feature, because the ultimate goal of a computational model is to incorporate all the architectural issues that are fundamental to the development of parallel applications, hopefully to the point that parallel machines can be designed according to the primitives of the model.

In the remaining chapters of the thesis, we have presented original results which are applications of structured parallelism as modeled by D-BSP. In Chapter 3 we have shown how to leverage on network locality to obtain an efficient implementation of a shared memory abstraction. The deterministic scheme we have devised generalizes the one in [PPS00] by making nontrivial modifications to the memory organization and by expressing the access protocol in terms of a few general primitives, so that it can adapt to the hierarchical structure of different architectures. As a result, the scheme can be immediately ported to any architecture supporting D-BSP. For the sake of concreteness, in the chapter we have analyzed the slowdown for a spectrum of D-BSP parameters, showing that close to optimal performance can be obtained for machines characterized by moderate bandwidth, such as multidimensional arrays. As a corollary, the instantiation of the scheme on the mesh yields the results in [PPS00]. Moreover, we have shown that optimality can be achieved on machines where delays due to latency dominate over those due to bandwidth limitations: none of the schemes previously developed in the literature for specific networks had revealed this fact. Common to all previous works on deterministic PRAM simulation, a challenging open problem is the explicit construction of expanding bipartite graphs that could make the scheme fully constructive for any number m of shared variables. As argued in the chapter, explicit constructions are known only to deal with the case $m = O\left(n^{3/2}\right)$, which however is sufficient to simulate any NC algorithm [JáJ92].

In Chapter 4, we have proved that temporal locality and network locality as modeled by D-BSP can be described in a unified framework and can be exploited through common strategies; the result is based on cross simulations of D-BSP machines. More precisely, we have shown how a guest D-BSP machine with v processors can be simulated on a v'-processor host configuration, $v' \leq v$, with slowdown proportional to v/v'. Since the original D-BSP does not model temporal locality, as a prerequisite towards our result we have extended D-BSP by making each node an HMM machine [AACS87]; furthermore, we regarded the cost of a network access in a cluster equal to the cost of a memory access to the farthest cell in the cluster. This choice is completely reasonable, since it indicates that the network is considered as the slowest level of the memory hierarchy. The significance of our result stems from the observation that, if $v' < v$, the local memories of the host machine are bigger (thus slower) than those of the guest machine, hence a *superlinear* slowdown could be likely. Although this expectation comes up to be true for highly local computations on point-to-point interconnection topologies [BP99], our result proves that it is not the case for bulk parallel computations. This is a remarkable finding, since bulk computations can be used to efficiently solve many prominent problems. To give evidence of this fact, our cross-simulation strategy has been applied to obtain optimal, sequential solutions for matrix multiplication, FFT and sorting: this application of our result shows that algorithmic techniques developed in the realm of parallelism can be easily applied to the development of hierarchy-conscious algorithms, a further evidence of a fundamental connection between network and temporal locality. In this context, D-BSP shifts its role and becomes a “virtual” model whose parameters are set to match the memory hierarchy of the host machine.

We remark that not all problems are expected to interact with the memory hierarchy in the same way. As a matter of fact, some problems (among which the ones analyzed by ourselves and by other authors in the literature) are known to be solvable on different hierarchies with the same sequential algorithm [FLPR99], but computations exist that require different algorithms on different hierarchies [BP01]. As a whole, the fundamental question of which properties make a problem efficiently solvable in a hierarchical setting is still open and far to be solved. With respect to the framework we have introduced in Chapter 4, it would therefore be useful to evaluate the performance of the simulation on a wider set of problems. Indeed, it would be interesting to identify classes of problems which require different algorithms on different hierarchies, or even problems for which no hierarchy-efficient algorithms exist, since they could give insight on the features connected to such ill behavior. Our framework makes it natural to start the search by considering the class of *P-complete* problems [Joh90], that is, the class of polynomial-time problems for which no efficient *parallel* solution is known. Note also that, by our framework, finding a hierarchy-inefficient problem $\mathcal{P}$ immediately gives a significant lower bound on the parallel execution time for $\mathcal{P}$.

Building on the promising results of Chapter 4, in Chapter 5 we have made an effort to incorporate spatial locality of reference into our framework. In doing so, we have been pushed to develop a further extension of D-BSP which is capable of modeling temporal, spatial and network locality according to a single cost function $f(x)$; our extension minimizes the burden connected with a new model by resorting to block transfer facilities as described in the previous literature [ACS87, BDP99]. Through cross-simulations, we have proved that a Brent-like lemma still holds for the new model: in other words, a v-processor machine can be simulated on a v'-processor one having the same cost function and aggregate memory size with slow-down proportional to v/v'. This result, which may lead to think of a transformation of parallelism into spatial locality, gives only a limited view on the effects of spatial locality itself. In fact, by taking maximum advantage of block transfer we have been able to develop a more sophisticated simulation scheme that translates a D-BSP program into a sequential $f(x)$-BT program whose running time is *independent* of $f(x)$. This simulation scheme uncovers phenomena we did not observe before, which take us a step ahead of the seamless integration setting. Indeed, both our findings and results in the literature [ACS87] show that the main effect of spatial locality is to make algorithms mostly insensitive to $f(x)$, thus "flattening" the memory and network hierarchies. This is the true nature of the block transfer mechanism: since the cost of transferring a block of l data is $f(x) + l$, if l is big enough (viz., if $l \geq f(x)$ in this cost model) then the average cost per datum is $O(1)$. Therefore, an algorithm works as if run on a flat hierarchy if it is able to move sufficiently big chunks of data: on such a hierarchy, the exploitation of network and temporal locality becomes much less critical.

We remark that these optimal conditions are difficult to be met, because the problem at hand may not expose a sufficient degree of spatial locality: in fact, some algorithmic test cases in this chapter demonstrate that the presence of other forms of locality still improves the running time. Furthermore, it must be noticed that the considerations stated above crucially relies on the availability of *unlimited* block transfer, that is, a block transfer mechanism which is capable of transferring blocks of arbitrary size. Such a powerful device is justified by the nature of our study in the chapter, which aims at evaluating the maximum benefit which can be reaped from block transfer, but is in contrast with the vast body of work on 2-level memories where the block size B is a parameter of the model. Unlimited block transfer is not present in current memory implementations, although its feasibility has recently been advocated [BEP02], while it is available in high-performance networks in the form of *wormhole routing*.

As a whole, it is not clear whether unlimited block transfer is a reasonable candidate for inclusion in a bridging model. For instance, consider a "crippled" $f(x)$-BT machine where the block size depends on the memory hierarchy level to be accessed: to be precise, the crippled model is able to copy a memory region $[x-l, x]$ into a non-overlapping region $[y-l, y]$ in chunks of at most $\max\{f(x), f(y)\}$ cells. It can be easily seen that such a restricted model is able to simulate an arbitrary $f(x)$-BT program with constant slowdown, hence unlimited block transfer is not strictly necessary to attain maximum performance. This observation suggests that more work on models needs to be done: for instance, it would be interesting to investigate more limited forms of block transfer within our framework. With respect to models, we also feel that seamless integration is not to be abandoned altogether, since a model which deals with the different manifestations of locality in a uniform fashion would be undoubtedly useful.

List of Figures

List of Tables

Bibliography

[AACS87] Alok Aggarwal, Bowen Alpern, Ashok K. Chandra, and Marc Snir. A model for hierarchical memory. In *Proceedings of the 19th Annual ACM Symposium on the Theory of Computing*, pages 305–314, 1987.

[ACF93] Bowen Alpern, Larry Carter, and Jeanne Ferrante. Modeling parallel computers as memory hierarchies. In W. K. Gilio, S. Jähnichen, and B. D. Shriver, editors, *Programming Models for Massively Parallel Computers*, pages 116–123. IEEE Computer Society Press, 1993.

[ACS87] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Hierarchical memory with block transfer. In *Proceedings of 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, Los Angeles, USA, October 1987.

[ACS89] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. On communication latency in PRAM computations. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21, Santa Fe, New Mexico, USA, June 1989. ACM Press.

[AHMP87] Helmut Alt, Torben Hagerup, Kurt Mehlhorn, and Franco P. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal on Computing*, 16(5):808–835, 1987.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the 15th Annual ACM Symposium on the Theory of Computing*, pages 1–9, Boston, Massachusetts, USA, April 1983.

[APM$^+$00] Nancy M. Amato, Jack Perdue, Mark M. Mathis, Andrea Pietracaprina, and Geppino Pucci. Predicting performance on SMPs. a case study: The SGI Power Challenge. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 729–737, Cancun, Mexico, May 2000. IEEE Computer Society.

[BDMadH98] Armin Bäumker, Wolfgang Dittrich, and Friedhelm Meyer auf der Heide. Truly efficient parallel algorithms: 1-optimal multisearch for an extension of the BSP model. *Theoretical Computer Science*, 203:175–203, 1998.

[BDP99] Armin Bäumker, Wolfgang Dittrich, and Andrea Pietracaprina. The complexity of parallel multisearch on coarse-grained machines. *Algorithmica*, 24:209–242, 1999.

[BEP02] Gianfranco Bilardi, Kattamuri Ekanadham, and Pratap Pattnaik. Optimal organizations for pipelined hierarchical memories. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 109–116, Winnipeg, Manitoba, Canada, August 2002. ACM Press.

[BFPP01] Gianfranco Bilardi, Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci. On the effectiveness of D-BSP as a bridging model of parallel computation. In *Proceedings of ICCS 2001*, LNCS 2074, pages 579–588, San Francisco, California, USA, May 2001.

[BGMZ97] Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Marco Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):943–958, 1997.

[BHP$^+$96] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul G. Spirakis. BSP vs LogP. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 25–32, Padova, Italy, June 1996. ACM Press.

[BHPP00] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, and Geppino Pucci. On stalling in LogP. In *Proceedings of IPDPS 2000 Workshops*, pages 109–115, Cancun, Mexico, May 2000. Springer Verlag.

[BP97] Gianfranco Bilardi and Franco P. Preparata. Processor-time tradeoffs under bounded-speed message propagation: Part I, upper bounds. *Theory of Computing Systems*, 30:523–546, 1997.

[BP99] Gianfranco Bilardi and Franco P. Preparata. Processor-time tradeoffs under bounded-speed message propagation: Part II, lower bounds. *Theory of Computing Systems*, 32:531–559, 1999.

[BP00] Mauro Bianco and Geppino Pucci. On the predictive quality of BSP-like cost functions for NOWs. In *Proceedings of Euro-Par 2000*, LNCS 1900, pages 638–646, Munich, Germany, August 2000. Springer Verlag.

[BP01] Gianfranco Bilardi and Enoch Peserico. A characterization of temporal locality and its portability across memory hierarchies. In *Proceedings of ICALP 2001*, LNCS 2076, pages 128–139, 2001.

[BPP99] Gianfranco Bilardi, Andrea Pietracaprina, and Geppino Pucci. A quantitative measure of portability with application to bandwidth-latency models for parallel computing. In *Proceedings of Euro-Par 99*, LNCS 1685, pages 543–551, Toulouse, France, August 1999.

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[CGG$^+$95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, San Francisco, California, USA, January 1995.

[CKP$^+$96] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Eunice E. Santos, Klaus E. Schauser, Ramesh Subramonian, and Thorsten von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

[CMadHS00] Artur Czumaj, Friedhelm Meyer auf der Heide, and Volker Stemann. Contention resolution in hashing based shared memory simulations. *SIAM Journal on Computing*, 29(5):1703–1739, 2000.

[CT65] James C. Cooley and John W. Tukey. An algorithm for the machine computation of the complex Fourier series. *Mathematics of Computation*, 19:297–301, April 1965.

[CW79] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, April 1979.

[DDH03] Frank Dehne, Wolfgang Dittrich, and David Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. *Algorithmica*, 36(2):97–122, April 2003.

[DDHM99] Frank Dehne, Wolfgang Dittrich, David Hutchinson, and Anil Maheshwari. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proceedings of the 13th International Parallel Processing Symposium*, pages 14–20, 1999.

[DFRC96] Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400, 1996.

[dK96] Pilar de la Torre and Clyde P. Kruskal. Submachine locality in the bulk synchronous setting. In *Proceedings of Euro-Par 96*, LNCS 1124, pages 352–358, Lyon, France, August 1996.

[FLPR99] Matteo Frigo, Charles Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 285–297, October 1999.

[FPP] Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci. Translating submachine locality into locality of reference. Submitted to IPDPS 2004.

[FPP01] Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci. Implementing shared memory on clustered machines. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, San Francisco, California, USA, April 2001. IEEE Computer Society.

[FPP02] Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci. Seamless integration of parallelism and memory hierarchy. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, LNCS 2380, pages 856–867, Malaga, Spain, July 2002. Springer Verlag.

[FPP03] Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci. A general PRAM simulation scheme for clustered machines. *International Journal on Foundations of Computer Science*, 2003. To appear.

[FW78] Steve Fortune and James Wyllie. Parallelism in random access machines. In *Conference Record of the 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, San Diego, California, USA, May 1978.

[GLR$^+$96] Mark Goudreau, Kevin Lang, Satish Rao, Torsten Suel, and Thanasis Tsantilas. Towards efficiency and portability: Programming with the

BSP model. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, Padova, Italy, June 1996. ACM Press.

[GMR99] Leslie Ann Goldberg, Yossi Matias, and Satish Rao. An optical simulation of shared memory. *SIAM Journal on Computing*, 28(5):1829–1847, 1999.

[Goo93] Michael T. Goodrich. Parallel algorithms column I: Models of computation. *SIGACT News*, 24(4):16–21, December 1993.

[Goo96] Michael T. Goodrich. Communication-efficient parallel sorting. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, pages 247–256, Philadelphia, Pennsylvania, USA, May 1996. ACM Press.

[HB94] Kieran T. Herley and Gianfranco Bilardi. Deterministic simulations of PRAMs on bounded-degree networks. *SIAM Journal on Computing*, 23(2):276–292, 1994.

[Her96] Kieran T. Herley. Representing shared data in distributed-memory parallel computers. *Mathematical Systems Theory*, 29(2):111–156, 1996.

[HJ86] Marshall Hall Jr. *Combinatorial Theory*. John Wiley & Sons, New York, NY, second edition, 1986.

[HL95] Todd Heywood and Claudia Leopold. *Models of Parallelism*, chapter 1-16. Oxford University Press, 1995.

[HPP01] Kieran T. Herley, Andrea Pietracaprina, and Geppino Pucci. Implementing shared memory on mesh-connected computers and on the fat-tree. *Information and Computation*, 165(2):123–143, 2001.

[HR92] Todd Heywood and Sanjay Ranka. A practical hierarchical model of parallel computation. I. the model. *Journal of Parallel and Distributed Computing*, 16(3):212–232, 1992.

[JáJ92] Joseph F. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, Reading, MA, 1992.

[Joh90] David S. Johnson. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, chapter "A Catalog of Complexity Classes", pages 67–161. Elsevier and MIT Press, 1990.

[JW94] Ben H. H. Juurlink and Harry A. G. Wijshoff. The parallel hierarchical memory model. In *Proceedings of the 4th Scandinavian Workshop on Algorithm Theory*, LNCS 824, pages 240–251, Aarhus, Denmark, July 1994. Springer Verlag.

[JW96] Ben H. H. Juurlink and Harry A. G. Wijshoff. The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Proceedings of Euro-Par 96, Volume II*, pages 339–347, Lyon, France, August 1996. Springer Verlag.

[JW98] Ben H. H. Juurlink and Harry A. G. Wijshoff. A quantitative comparison of parallel computation models. *ACM Transactions on Computer Systems*, 16(3):271–318, August 1998.

[Lei85] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, October 1985.

[Lei92] Frank Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes.* Morgan Kaufmann, San Mateo, CA, 1992.

[LMR99] Frank Thomson Leighton, Bruce M. Maggs, and Andréa W. Richa. Fast algorithms for finding O(congestion + dilation) packet routing schedules. *Combinatorica*, 19(3):375–401, 1999.

[LMRR94] Frank Thomson Leighton, Bruce M. Maggs, Abhiram G. Ranade, and Satish Rao. Randomized routing and sorting on fixed-connection networks. *Journal of Algorithms*, 17(1):157–205, July 1994.

[LP93] Fabrizio Luccio and Linda Pagli. A model of sequential computation with pipelined access to memory. *Mathematical Systems Theory*, 26(4):343–356, 1993.

[LPP90] Fabrizio Luccio, Andrea Pietracaprina, and Geppino Pucci. A new scheme for the deterministic simulation of PRAM in VLSI. *Algorithmica*, 5:529–544, 1990.

[LR99] Frank Thomson Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999.

[Mil94] R. Miller. *Two approaches to architecture-independent parallel computation.* PhD thesis, Oxford University, 1994.

[MMT95] Bruce M. Maggs, Lesley R. Matheson, and Robert Endre Tarjan. Models of parallel computation: a survey and synthesis. In *Proccedings of the 28th Annual Hawaii International Conference on System Sciences*, pages 61–72, Kihei, Maui, Hawaii, USA, January 1995. IEEE Computer Society Press.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[MV84] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[PP97] Andrea Pietracaprina and Franco P. Preparata. Practical constructive schemes for deterministic shared-memory access. *Theory of Computing Systems*, 33:3–37, 1997.

[PPA$^+$98] Andrea Pietracaprina, Geppino Pucci, Nancy M. Amato, Lucia K. Dale, and Jack Perdue. A cost model for communication on a symmetric multiprocessor. In *10th ACM Symposium on Parallel Algorithms and Architectures, Revue Session*, Puerto Vallarta, Mexico, June-July 1998.

[PPS00] Andrea Pietracaprina, Geppino Pucci, and Jop F. Sibeyn. Constructive, deterministic implementation of shared memory on meshes. *SIAM Journal on Computing*, 30(2):625–648, 2000.

[Ran91] Abhiram G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42:307–326, 1991.

[Reg96] Kenneth W. Regan. Linear time and memory-efficient computation. *SIAM Journal on Computing*, 25(1):133–168, February 1996.

[SA$^+$92] H. J. Siegel, S. Abraham, et al. The Purdue workshop on grand challenges in computer architecture for the support of high performance computing. *Journal of Parallel and Distributed Computing*, 16(3):199–211, November 1992.

[SK94] Jop F. Sibeyn and Michael Kaufmann. Deterministic 1-k routing on meshes. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science*, LNCS 775, pages 237–248, Caen, France, February 1994. Springer Verlag.

[SK97] Jop F. Sibeyn and Michael Kaufmann. BSP-like external-memory computation. In *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, LNCS 1203, pages 229–240, 1997.

[UW87] Eli Upfal and Avi Widgerson. How to share memory in a distributed system. *Journal of the ACM*, 34(1):116–127, 1987.

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

[Val90b] Leslie G. Valiant. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, chapter "General Purpose Parallel Architectures", pages 943–972. Elsevier and MIT Press, 1990.

[Vis96] Uzi Vishkin. Can parallel algorithms enhance serial implementation? *Communications of the ACM*, 39(9):88–91, 1996.

[Vit91] Jeffrey Scott Vitter. Efficient memory access in large-scale computation. In *Proceedings of the 8th Annual Symposium on Theoretical Aspects of Computer Science*, LNCS 480, pages 26–41, Hamburg, Germany, February 1991. Springer Verlag.

[VS94a] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[VS94b] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2-3):148–169, 1994.