Translating Submachine Locality into Locality of Reference ⋆

Carlo Fantozzi Andrea Pietracaprina Geppino Pucci *

Dipartimento di Ingegneria dell'Informazione, Università di Padova, Padova, Italy

Abstract

In this work, we show that the submachine locality exposed by hierarchical bulk-synchronous computations can be efficiently turned into locality of reference on arbitrarily deep hierarchies. Specifically, we develop efficient schemes to simulate parallel programs written for the Decomposable BSP (a BSP variant which features a hierarchical decomposition into submachines) on the sequential Hierarchical Memory Model (HMM), which rewards the exploitation of temporal locality, and on its extension with block transfer, the BT model, which also rewards the exploitation of spatial locality. The simulations yield 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 in the hierarchical memory setting. We also devise a generalization of the HMM result to the self-simulation of D-BSP augmented with hierarchical memory modules, which yields an interesting analogue of Brent's lemma, thus proving that the enhanced model features a seamless integration of memory and network hierarchies.

Key words: Decomposable BSP, Hierarchical Memory Model, Block Transfer, Submachine Locality, Locality of Reference

⋆ This research was supported in part by MIUR of Italy under project "ALGO-NEXT: ALGOrithms for the NEXT generation Internet and the Web", and by the University of Padova under Grant CPDA033838. Preliminary versions of the results in the paper were presented at the *18th International Parallel and Distributed Processing Symposium* (IPDPS 2004, Best Paper Award, *Algorithms* Track), and at the *29th International Colloquium on Automata, Languages, and Programming* (ICALP 2002).

* **Corresponding Author**: Dip. di Ingegneria dell'Informazione, Via Gradenigo 6/B, 35131 Padova, Italy. Tel. +39 0498277951, Fax: +39 0498277799

Email addresses: carlo.fantozzi@unipd.it (Carlo Fantozzi), andrea.pietracaprina@unipd.it (Andrea Pietracaprina), geppino.pucci@unipd.it (Geppino Pucci).

1 Introduction

In modern computing platforms the memory system is characterized by a multilevel hierarchical structure, with capacity and access times increasing as levels grow farther from the processing unit. In multiprocessors the communication network introduces further levels in the hierarchy. Since the time to access data varies with the level being accessed, performance is considerably enhanced when the relevant data can be moved (possibly in blocks) up the hierarchy, close to the unit (or the units) that processes it. To exploit this fact, the computation must exhibit a property generally referred to as data locality. On a uniprocessor, data locality, also known as locality of reference, takes two distinct forms, namely, the frequent reuse of data within short time intervals (*temporal locality*), and the access to consecutive data in subsequent operations (*spatial locality*). On multiprocessors, another form of locality is *submachine* or *communication locality*, which requires that data be distributed so that communications are confined within small submachines featuring high bandwidth and small latency.

It is well known that classical algorithms for prominent computational problems developed under the assumption of flat memory (RAM model) often exhibit poor performance when run on real machines with hierarchical memory. As a consequence, in the last decade, a number of models of sequential computation have been proposed that explicitly account for the hierarchical structure of the memory system. Among the others, the *Hierarchical Memory Model* (HMM) defined in [1], is a random access machine where access to memory location x requires time $f(x)$, for a given nondecreasing function f, thus encouraging the exploitation of temporal locality. The *Hierarchical Memory Model with Block Transfer* (BT, for short) [2,3] 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. Also, a two-level memory organization is featured by the *External Memory* (EM) model [4,5], which has been extensively used in the literature to develop I/O-efficient algorithms. Finally, other multilevel hierarchical models have been defined in [6,7].

Earlier works provided evidence that efficient sequential algorithms for two-level hierarchies can be obtained by simulating parallel ones. In [8–10] schemes are presented that simulate parallel algorithms designed for coarse-

grained parallel models, such as BSP [11], BSP* [12], CGM [13], 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 to obtain efficient algorithms on deeper hierarchies.

The simulation of PRAM algorithms to obtain efficient sequential ones was explored in [14], where parallelism is turned into efficient cache prefetching strategies. Another approach to the development of efficient hierarchy-conscious algorithms was proposed in [15], also 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.

A more general study on the interplay between communication locality of a parallel algorithm and temporal locality of reference is attempted by Bilardi and Preparata in [16]. The authors introduce the $M_d(n, p, m)$ model, 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. In order to evaluate how much communication locality can be automatically transformed into temporal locality, the authors resort to self-simulation results, showing 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 [17], 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 [18]. By setting $p = 1$, the simulation strategy provides HMM algorithms starting from parallel ones.

The superlinear slowdown result in [16,18] substantiates the widespread belief that the implementation of a parallel algorithm on a sequential machine may incur an extra slowdown over the mere loss of parallelism, due to the aggregation of the distinct processors' hierarchies into a single, deeper one. In this

paper, we complement the above result by showing that the submachine locality exposed by hierarchical bulk-synchronous computations can be profitably translated into locality of reference, proving that for such computations no extra hierarchy-induced slowdown is incurred when scaling down the number of processors. Our result is made significant by the fact that several prominent problems can be optimally solved through hierarchical bulk-synchronous computations.

Our study is based on the Decomposable BSP (D-BSP) model, a variant of BSP [11] introduced in [19] to account for submachine locality. Specifically, a v-processor D-BSP consists of a collection of v processors communicating through a router. The processors are partitioned into several levels of independent submachines (clusters) according to a binary decomposition tree for the machine. The main contribution of the paper is a scheme that simulates any v-processor D-BSP computation on a sequential HMM machine with the same aggregate memory size. When the cost of communication within a D-BSP cluster is chosen to be proportional to the cost of accessing the farthest cell in an HMM memory of size equal to the cluster's aggregate memory, our simulation exhibits an optimal, linear slowdown merely proportional to the loss of parallelism. 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 [20] for porting DAG computations efficiently across sequential memory hierarchies.

The above 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 (e.g., sorting, FFT and matrix multiplication), optimal HMM algorithms can be obtained in this fashion. In this respect, our work provides a generalization of the results in [8,9] to multi-level memory hierarchies.

Next, we show how the simulation scheme can be adapted to map any v-processor D-BSP computation on a v'-processor D-BSP, with $v' \leq v$, where both machines feature the same amount of aggregate memory and individual processors are regarded as sequential HMMs. The simulation exhibits an optimal $O\left(v/v'\right)$ slowdown, thus yielding an analogue of Brent's lemma. This result

provides evidence that the D-BSP model, when augmented with hierarchical memory modules, successfully integrates memory and network hierarchies by regarding the latter as a seamless continuation of the former.

Finally, we extend the investigation to encompass spatial locality, so to assess to what extent submachine locality can lead to a combined exploitation of both forms of locality of reference in multi-level hierarchies. More specifically, we turn the HMM simulation of D-BSP into a simulation of D-BSP on the BT model. This latter simulation reveals that optimal or quasi-optimal BT algorithms can be obtained starting from D-BSP algorithms exhibiting a much coarser level of submachine locality than the one needed for the HMM, which strictly depends on the access function. Our results are in accordance with those in [2], 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 locality of communications 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 powerful tool to obtain efficient hierarchy-conscious algorithms automatically from the large body of parallel algorithms developed in the literature over the last two decades.

The rest of the paper is organized as follows. Section 2 defines our reference models. Section 3 presents the scheme that simulates an arbitrary D-BSP computation on the HMM, and shows how optimal HMM algorithms can be obtained from optimal parallel ones. The generalized D-BSP self-simulation yielding the analogue of Brent's Lemma and the related modeling issues are discussed in Section 4. The extension of the simulation to the BT model is presented and analyzed in Section 5. Finally, Section 6 offers a number of concluding remarks.

2 Machine Models

HMM The *Hierarchical Memory Model* (HMM) was introduced in [1] as a random access machine where access to memory location x requires time $f(x)$, for a given nondecreasing function $f(x)$. The model assumes that an n-ary 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 will focus our attention on nondecreasing functions $f(x)$ for which there exists a constant $c \geq 1$ such that $f(2x) \leq cf(x)$, for any x. As in [20], we will refer to these functions as $(2, c)$-*uniform* (in the literature these functions have also been called *well behaved* [3] or, somewhat improperly, *polynomially bounded* [1]). 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 technical fact is proved in [1].

Fact 1 *If $f(x)$ is $(2, c)$-uniform, then the time to access the first n memory cells of an $f(x)$-HMM is $\Theta(nf(n))$.*

BT The *Hierarchical Memory Model with Block Transfer* was introduced in [2] by augmenting the $f(x)$-HMM model with a block transfer facility. We refer to this model as $f(x)$-BT. Specifically, as in the $f(x)$-HMM, an access to memory location x requires time $f(x)$, but the model makes also possible to copy a block of b memory cells $[x - b + 1, x]$ into a *disjoint* block $[y - b + 1, y]$ in time $\max\{f(x), f(y)\} + b$, for arbitrary $b > 1$. As before, we will restrict our attention to the case of $(2, c)$-uniform access functions.

It must be remarked that the block transfer mechanism featured by the model is rather 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. Let $f^{(k)}(x)$ be the iterated function obtained by applying function f k times, and let $f^*(x) = \min\{k \geq 1 : f^{(k)}(x) \leq 1\}$. The following fact is easily established from [2].

Fact 2 *The touching problem on $f(x)$-BT requires time $T_{\mathrm{TCH}}(n) = \Theta(nf^*(n))$. In particular, we have that $T_{\mathrm{TCH}}(n) = \Theta(n \log^* n)$ if $f(x) = \log x$, and $T_{\mathrm{TCH}}(n) = \Theta(n \log \log n)$ if $f(x) = x^{\alpha}$, for a positive constant $\alpha < 1$.*

The fact 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.

The memory transfer capabilities postulated by the BT model are al-

ready (reasonably) well approximated by current hierarchical designs. First of all observe that $f(x)$-BT can be simulated with constant slowdown by a restricted version of the model which in time $f(x)$ allows only to transfer $f(x)$ consecutive cells between non-overlapping regions of maximum address x. In a current generation machine, the latency $f(x)$ for an access to main memory is in the order of $100 \div 200$ cycles; considering that line sizes of $8 \div 16$ words and about 10 simultaneously outstanding memory requests are typical, in the aforementioned $100 \div 200$ cycles about $80 \div 160$ words can be accessed, which meets the restricted BT capabilities to an excellent degree. The situation is similar for other levels of the internal and external hierarchy. Furthermore, in [21] the physical and architectural feasibility has been established, in principle, for hierarchical memories where even non-consecutive addresses can be pipelined, thus leading to a model even more powerful than BT (where, for example, touching can be accomplished in linear time).

D-BSP The *Decomposable Bulk Synchronous Parallel* (D-BSP) model was introduced in [19] to capture submachine locality in a structured way through submachine decomposition, and was further investigated in [22–24].

Let v be a power of two. A D-BSP$(v, \mu, g(x))$ is a collection of v processors $\{P_j : 0 \leq j < v\}$ communicating through a router whose bandwidth characteristics are captured by function $g(x)$; each processor is equipped with a local memory of size μ. For $0 \leq i \leq \log v$, the v processors are partitioned into 2^i fixed, disjoint *i-clusters* $C_0^{(i)}, C_1^{(i)}, \cdots, C_{2^i-1}^{(i)}$ of $v/2^i$ processors each, where 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: specifically, $C_j^{\log v} = \{P_j\}$, for $0 \leq j < v$, and $C_j^{(i)} = C_{2j}^{(i+1)} \cup C_{2j+1}^{(i+1)}$, for $0 \leq i < \log v$ and $0 \leq j < 2^i$.

A D-BSP program consists of a sequence of *labeled supersteps*. In an *i-superstep*, $0 \leq i \leq \log v$, each processor executes internal computation on locally held data and sends messages exclusively to processors within its i-cluster (buffers for incoming and outgoing messages are provided as part of the processor's local memory). 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 next superstep. It is also reasonable to assume that any D-BSP computation ends with a global

synchronization, i.e., a 0-superstep. 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, $h > 0$, (i.e., each processor is the source or destination of at most h messages), then the cost of the i-superstep is upper bounded by $\tau + hg(\mu v/2^i)$. Note that since message buffers are part of the processors' contexts h cannot be larger than μ. With this particular choice of the cost function, each message delivery performed in an i-superstep is envisioned as a sort of remote access with access function $g(x)$ outside the aggregate memory of an i-cluster.

Since our main objective is to assess to what extent submachine locality can be transformed into locality of reference, in the paper we will mostly be concerned with the simulation of *fine-grained* D-BSP programs where the local memory of each processor has constant size (i.e., $\mu = O(1)$). In this fashion, submachine locality is the only locality that can be exhibited by the parallel program. We will deviate from this scenario only in Section 4 where, in order to obtain self-simulations of D-BSP machines, we will need the individual D-BSP processors to feature internal hierarchies of nonconstant size. (For clarity, throughout the paper we will always indicate μ in the asymptotic expressions even in the case of fine-grained programs.)

Although in this paper we deal with arbitrary $(2, c)$-uniform access functions $f(x)$, we will support our findings by considering, as case studies, the aforementioned polynomial and logarithmic functions.

3 Simulation of D-BSP on HMM

The core result of this section (Theorem 5 and Corollary 6) is a scheme to simulate a fine-grained D-BSP$(v, \mu, g(x))$ program $\mathcal{P}$ on an $f(x)$-HMM with $(2, c)$-uniform access function f, with a slowdown which is merely proportional to the loss of parallelism (i.e., slowdown linear in v). The scheme succeeds in hiding the memory hierarchy costs induced by the HMM access function by efficiently transforming submachine locality into temporal locality of reference.

We refer to D-BSP and HMM as the *guest* and *host* machine, respectively. The memory of the host machine is divided into *blocks* of μ cells each, with block 0 at the top of memory. At the beginning of the simulation, block j, $j = 0, 1, \ldots, v-1$, contains the *context* (i.e., the local memory) of processor P_j, but this association changes as the simulation proceeds.

For technical purposes, we need to define the following special class of D-BSP$(v, \mu, g(x))$ programs.

Definition 3 *Let $\mathcal{L}$ be a fixed subset of superstep labels $0 = \ell_0 < \ell_1 < \ldots < \ell_m = \log v$. A program $\mathcal{P}$ for a D-BSP$(v, \mu, g(x))$ is* $\mathcal{L}$-smooth *if the following two properties hold:*

(1) The label of every superstep of $\mathcal{P}$ belongs to $\mathcal{L}$;
(2) If a superstep with label ℓ_i directly follows a superstep with label $\ell_j > \ell_i$ then $i = j - 1$.

For convenience, we assume that the input D-BSP program $\mathcal{P}$ is $\mathcal{L}$-smooth, for some set of indices $\mathcal{L}$. If this is not the case, $\mathcal{P}$ can be easily turned into a functionally equivalent $\mathcal{L}$-smooth program $\mathcal{P}'$ by performing the following two operations in sequence.

(1) First, "upgrade" each i-superstep in $\mathcal{P}$ to an ℓ-superstep, with ℓ the largest index in $\mathcal{L}$ not greater than i. Note that in this fashion several supersteps of different indices may be bundled into a single group of supersteps with the same index.
(2) Then, add a suitable number of "dummy" supersteps with the missing labels in $\mathcal{L}$ between consecutive supersteps pairs (ℓ_j, ℓ_i) such that $i < j-1$.

Although this transformation may yield a higher D-BSP running time for $\mathcal{P}'$, our analysis will study the slowdown of the simulation with respect to the running time of the original program $\mathcal{P}$.

Let the supersteps of $\mathcal{P}'$ be numbered consecutively, and let i_s be the label of the s-th superstep, with $s \geq 0$ (i.e., the s-th superstep is executed independently within i_s-clusters). At some arbitrary point during the simulation, an i_s-cluster C is said to be *s-ready* if, for all processors in C, supersteps $0, 1, \ldots, s-1$ have been simulated, while Superstep s has not been simulated yet. The simulation, whose pseudocode is given in Figure 1, is organized into a number of *rounds*, corresponding to the iterations of the while loop in the code. A round simulates the operations prescribed by a certain Superstep s for a certain s-ready cluster C, and performs a number of context swaps to prepare for the execution of the next round. In order to transform the submachine locality exhibited by the D-BSP program into temporal locality on the HMM, the simulation proceeds unevenly on the different D-BSP clusters. This is achieved by suitably selecting the next cluster to be simulated after each round, which, if needed, must be brought on top of memory. In particular,

```
    while true do
1       P ← processor whose context is on top of memory
        s ← superstep number to be simulated next for P
        C ← i_s-cluster containing P
2       Simulate Superstep s for C
3       if P has finished its program then exit
4       if i_{s+1} < i_s then
            b ← 2^{i_s − i_{s+1}}
            Let Ĉ be the i_{s+1}-cluster containing C,
                and let Ĉ_0 ... Ĉ_{b−1} be its component i_s-clusters,
                with C = Ĉ_j for some index j
            if j > 0 then swap the contexts of C with those of Ĉ_0
            if j < b − 1 then
                swap the contexts of Ĉ_0 with those of Ĉ_{j+1}
```

Fig. 1. The simulation algorithm

the same cluster could be simulated for several consecutive supersteps so to avoid repeated, expensive relocations of its processors' contexts in memory. Note that the simulation is *on-line* in the sense that the entire sequence of supersteps needs not be known by the processors in advance. Moreover, the simulation code is totally oblivious to the D-BSP bandwidth function $g(x)$.

As will be proved later, the algorithm maintains the following two invariants at the beginning of each round. Let s and C be defined as in Step 1 of the round.

Invariant 1 *C is s-ready.*

Invariant 2 *The contexts of all processors in C are stored in the topmost $|C|$ blocks, sorted in increasing order by processor number. Moreover, for any other cluster C', the contexts of all processors in C' are stored in consecutive memory blocks (although not necessarily sorted).*

Consider a generic round where Superstep s is simulated for an i_s-cluster C. If $i_{s+1} \geq i_s$ then no cluster swaps are performed at the end of the round, and the next round will simulate Superstep $s+1$ for the topmost i_{s+1}-cluster contained in C and currently residing on top of memory. Such a cluster is clearly $(s+1)$-ready. Instead, if $i_{s+1} < i_s$, Superstep $s+1$ involves a coarser level of clustering, hence the simulation of this superstep can take place only after Superstep s has been simulated for *all* i_s-clusters that form the i_{s+1}-cluster $\hat{C}$ containing C. Step 4 is designed to enforce this schedule. In particular, let $\hat{C}$ contain $b = 2^{i_s - i_{s+1}}$ i_s-clusters, including C, which we denote by $\hat{C}_0, \hat{C}_1, \ldots, \hat{C}_{b-1}$, and suppose that $C = \hat{C}_0$ is the first such i_s-cluster for which

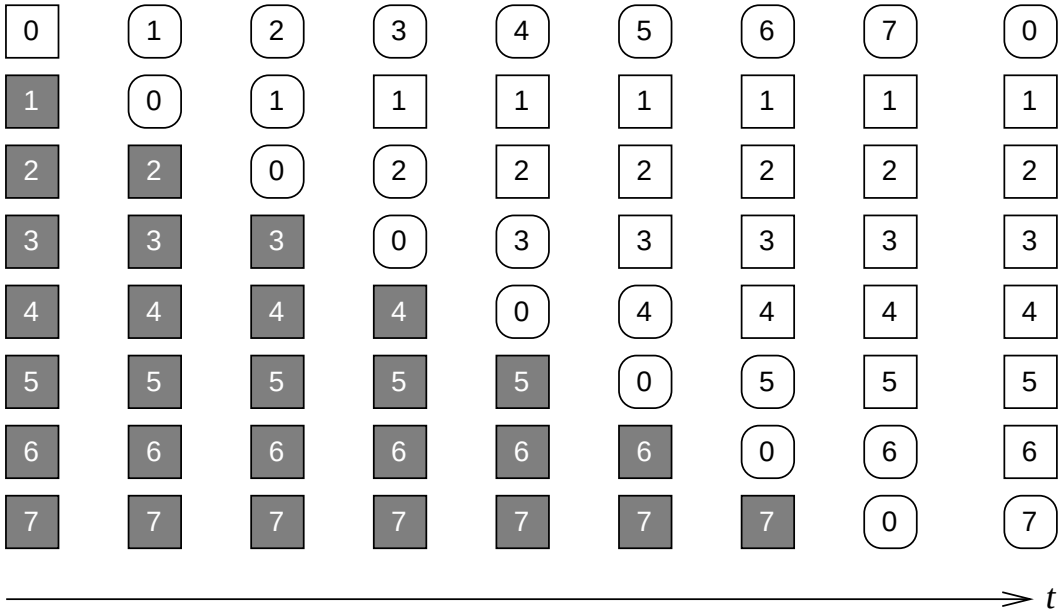

Fig. 2. Snapshots of the HMM memory highlighting cluster movements during a cycle involving an i_{s+1}-cluster that contains $b = 8$ i_s-clusters. Each box represents the processor contexts of a different i_s-cluster. Grey boxes indicate s-ready (i.e., not yet simulated) clusters, while white boxes refer to clusters that are being or have already been simulated. 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.

Superstep s is simulated. By Invariant 2, at the beginning of the round under consideration the contexts of all processors in $\hat{C}$ are at the top of memory. This round starts a *cycle* 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 moved back to the positions occupied at the beginning of the cycle. An example of the cluster movements performed during a cycle is depicted in Figure 2.

If the two invariants hold, then the simulation of cluster C in Step 2 can be performed as follows. First the context of each processor in C is brought in turn to the top of memory and its local computation is simulated. Then, message exchange is simulated by scanning the processors' outgoing message buffers sequentially and delivering each message to the incoming message buffer of the destination processor. The location of these buffers is easily determined since, by Invariant 2, the contexts of the processors are sorted by processor number.

Theorem 4 *The simulation algorithm is correct.*

PROOF. The correctness of the entire simulation algorithm is immediately established once we show that Invariants 1 and 2 hold at the beginning of each round. This can be proved by induction on the number v of D-BSP processors. The claim is trivial for the basis $v = 1$. Suppose that the claim is true for machines of up to v processors and consider the simulation of an $\mathcal{L}$-smooth program $\mathcal{P}'$ of t supersteps for a D-BSP with $2v$ processors. First, consider the case that 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 let $C_0, C_1, \ldots, C_{b-1}$, with $b = 2^{i_{t-1}}$, be the set of all i_{t-1}-clusters in the machine. It follows that the algorithm will initially simulate the first $t - 1$ supersteps of $\mathcal{P}'$ for the processors of cluster C_0, since the first $t - 1$ supersteps do not involve clusters larger than C_0 and $\mathcal{P}'$ is smooth. Moreover, these steps form themselves an $\mathcal{L}$-smooth program for a D-BSP with $2v/2^{i_{t-1}} \leq v$ processors.

By the inductive hypothesis, the two invariants will hold for all the rounds performed in this initial phase, which ends with a round that simulates Superstep $t - 1$ for cluster C_0. As observed before, at the end of such a round the algorithm proceeds with the remaining $b - 1$ phases that simulate the sibling i_{t-1}-clusters $C_1, C_2, \ldots, C_{b-1}$ by bringing each such cluster in turn to the top of memory and simulating up to the $(t - 1)$-th superstep for the processors of the cluster. After the simulation of Superstep $t - 1$ for C_{b-1} is completed, all i_{t-1}-clusters occupy their original positions, and the invariants hold at the beginning of the next final round, which will thus correctly simulate the 0-superstep.

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 such subprogram.

Let us now evaluate the running time of a generic round of our algorithm, where a given Superstep s is simulated for a given i_s-cluster C. Clearly, Steps 1 and 3 require constant time. Consider now Step 2, and note that the simulation of the local operations of each processor in C incurs constant slowdown, since it is done with the processor's (constant-size) context at the top of the HMM memory. Note also that by virtue of Invariant 2, message exchange can be completed by accessing the first $\mu|C|$ HMM memory cells only a constant number of times. Finally, it is easy to see that bringing the contexts iteratively

to the top of memory requires accessing the first $\mu|C|$ HMM memory cells only a constant number of times. Hence, letting τ_s denote the maximum local computation time for a processor in Superstep s, by Fact 1 the simulation of Step 2 is accomplished in time $O\left(|C|\left(\tau_s + \mu f(\mu|C|)\right)\right)$.

The running time of Step 4 is relevant only when $i_s > i_{s+1}$, and its analysis needs more careful consideration. We carry it out by aggregating the cost of all executions of Step 4 in those rounds where Superstep s is simulated for the $2^{i_s - i_{s+1}}$ i_s-clusters, including C, which form the i_{s+1}-cluster $\hat{C}$. In this cycle, memory blocks $0, 1, |C|-1$ are accessed a constant number of times for each i_s-cluster being simulated, so this cost is amortized by that of Step 2 in the simulation of that cluster. Memory blocks $|C|, |C|+1, \ldots, |\hat{C}|-1$ are read and written twice, hence the cost of these accesses is amortized by the cost of the future execution of Superstep $s+1$ for the $\hat{C}$-cluster.

By combining these observations and summing up the contributions of all the rounds in the simulation algorithm, we have that $\mathcal{P}'$ is simulated on $f(x)$-HMM in time

$$O\left(v\left(\tau + \mu \sum_{\ell \in \mathcal{L}} \lambda'_\ell f(\mu v/2^\ell)\right)\right), \tag{1}$$

where λ'_ℓ denotes the number of ℓ-supersteps executed by $\mathcal{P}'$, for $\ell \in \mathcal{L}$. However, recall that $\mathcal{P}'$ may represent the $\mathcal{L}$-smooth equivalent of an original program $\mathcal{P}$, and our goal is to obtain an expression for the simulation time dependent on the parameters of $\mathcal{P}$, rather than those of $\mathcal{P}'$. This goal can be achieved by smoothing $\mathcal{P}$ using a set of labels $\mathcal{L} = \{0 = \ell_0 < \ell_1 < \ldots < \ell_m = \log v\}$ such that, for two suitable fixed constants c_1, c_2, with $0 < c_1 \leq c_2 < 1$ it holds that: (a) $f(\mu v/2^{\ell_{i+1}}) \geq c_1 f(\mu v/2^{\ell_i})$, for every $0 \leq i < m$; and (b) $f(\mu v/2^{\ell_{i+1}}) \leq c_2 f(\mu v/2^{\ell_i})$, for every $0 \leq i < m-1$. $\mathcal{L}$ can be constructed iteratively as follows. Fix a constant $0 < c_2 < 1$ arbitrarily, and suppose that $0 = \ell_0 < \ell_1 < \ldots < \ell_i < \log v$ have already been determined. Then ℓ_{i+1} is chosen as the first index greater than ℓ_i such that $f(\mu v/2^{\ell_{i+1}}) \leq c_2 f(\mu v/2^{\ell_i})$, if any. If no such index exsists, then set $m = i+1$ and $\ell_m = \log v$. It is easy to see that since $f(x)$ is $(2, c)$-uniform it also holds that $f(\mu v/2^{\ell_{i+1}}) \geq c_1 f(\mu v/2^{\ell_i})$, with $c_1 = c_2/c \leq c_2$.

Consequently, each i-superstep of $\mathcal{P}$ is upgraded in $\mathcal{P}'$ to an ℓ-superstep, where i and ℓ are such that $f(\mu v/2^i) = \Theta\left(f(\mu v/2^\ell)\right)$. Moreover, in $\mathcal{P}'$ the total contribution of all the added dummy supersteps to Formula 1 cannot be

greater than a constant fraction of the one due to real supersteps. Thus, we have:

Theorem 5 *Consider a fine-grained D-BSP$(v, \mu, g(x))$ program $\mathcal{P}$, 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 $(2, c)$-uniform, then $\mathcal{P}$ can be simulated on a $f(x)$-HMM in time $O\left(v\left(\tau + \mu \sum_{i=0}^{\log v} \lambda_i f(\mu v/2^i)\right)\right)$.*

Tedious yet simple computations show that the hidden constant in the asymptotic expression of the simulation time is polynomial in the constant c. Such a dependence stems from the required smoothing of the program being simulated.

By setting the D-BSP$(v, \mu, g(x))$ bandwidth function $g(x)$ equal to the HMM access function $f(x)$, we obtain the following corollary which states the linear slowdown result claimed in the introduction.

Corollary 6 *If $f(x)$ is $(2, c)$-uniform then any T-time fine-grained program for a D-BSP$(v, \mu, f(x))$ can be simulated in optimal time $\Theta(Tv)$ on $f(x)$-HMM.*

3.1 Application to Case-Study Problems

In this section we focus on a number of prominent reference problems to show that our simulation can be employed to transform efficient fine-grained D-BSP algorithms into optimal solutions for those problems on the HMM. This provides evidence that the structured parallelism exposed in 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 obtain efficient, portable algorithms for hierarchical architectures.

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 [1]. 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

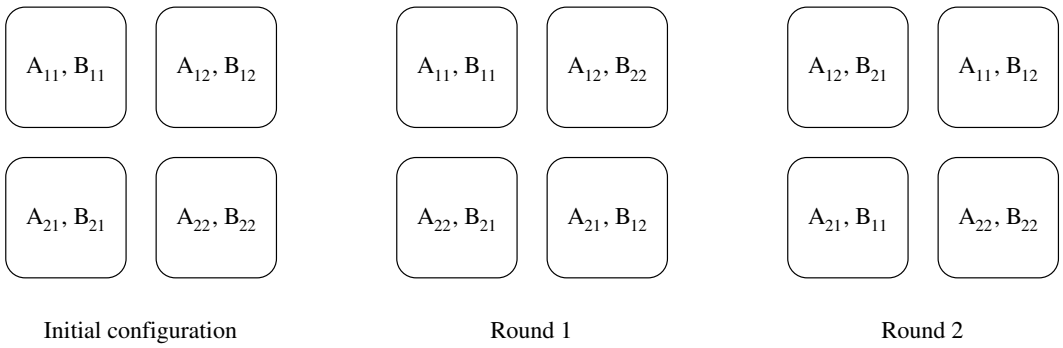

Fig. 3. Assignment of submatrices to the four D-BSP 2-clusters during the execution of the matrix multiplication algorithm

distributed among the D-BSP processors. We have

Proposition 7 *For the n-MM problem there is an algorithm that runs in time*

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

on D-BSP$(n, O(1), x^{\alpha})$, and in time $T^{\log}_{\mathrm{MM}}(n) = O\left(\sqrt{n}\right)$ on D-BSP$(n, O(1), \log x)$. The simulation of this algorithm yields optimal performance on x^{α}-HMM and $\log x$-HMM, respectively.

PROOF. We resort to the standard decomposition of n-MM into eight $(n/4)$-MM subproblems, which are then solved recursively by the four D-BSP 2-clusters. To keep memory space requirements at a minimum, the subproblems are solved in two rounds, organized in such a way that each submatrix is required exactly once in each round. A reasonable assignment of subproblems to the 2-clusters is sketched in Figure 3. In each round, the data can be moved to the correct position through one 0-superstep during which every D-BSP processor exchanges $O(1)$ data: the running time of the algorithm is therefore given by the recurrence equation

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

The solution of this equation leads to the stated D-BSP running times. By Corollary 6, the simulation of the algorithm on HMM yields a performance

matching the lower bounds for the n-MM problem proved in [1].

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 8 *For the n-DFT problem there is a D-BSP$(n, O(1), x^\alpha)$ algorithm that runs in time $T^\alpha_{\mathrm{DFT}}(n) = O(n^\alpha)$, and a D-BSP$(n, O(1), \log x)$ 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 x^α-HMM and $\log x$-HMM, respectively.*

PROOF. For D-BSP$(n, O(1), x^\alpha)$ it is sufficient to adopt the 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 its running time is

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

Instead, for D-BSP$(n, O(1), \log x)$ it is more efficient to recursively decompose the n-input FFT dag into 2 layers of $\sqrt{n}$ independent $\sqrt{n}$-input FFT subgraphs; the two layers are separated by a permutation, i.e., by a 0-superstep. Each subgraph can be scheduled onto a distinct $(\log n)/2$-cluster, hence the running time on D-BSP$(n, O(1), \log x)$ is given by the recurrence equation

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

which yields $T^{\log}_{\mathrm{DFT}}(n) = O(\log n \log\log n)$. Observe that the latter strategy would yield an alternative $O(n^\alpha)$-time algorithm for D-BSP$(n, O(1), x^\alpha)$.

The performance of the HMM algorithms obtained by applying the simulation result of Corollary 6 matches the best known bounds of $O(n^{1+\alpha})$ on x^α-HMM and of $O(n \log n \log\log n)$ on $\log x$-HMM, proved in [1].

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.

Proposition 9 *There is an n-sorting algorithm for D-BSP$(n, O(1), x^\alpha)$ that runs in time $T^\alpha_{\mathrm{SORT}}(n) = O(n^\alpha)$. The simulation of this algorithm on x^α-HMM exhibits optimal performance.*

PROOF. In [24, Proposition 2] a D-BSP$(n, O(1), x^\alpha)$ sorting algorithm is given that runs in time $O(n^\alpha)$. By Corollary 6, this algorithm can be simulated in time $O(n^{1+\alpha})$ on x^α-HMM. Optimality follows from the lower bound proved in [1].

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 on D-BSP$(n, O(1), \log x)$. 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 [1]. However, we conjecture that 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 $\Omega(\log n \log\log n)$ lower bound for n-sorting on the D-BSP$(n, O(1), \log x)$. No tighter lower bound or better algorithms are known so far.

4 An Analogue of Brent's Lemma

In order to extend the simulation result of Corollary 6 to the self-simulation of D-BSP machines, we need to handle the case of nonconstant local memory size μ. Specifically, we regard each D-BSP$(v, \mu, g(x))$ processor as a $g(x)$-HMM of size μ. Then, the following theorem provides an analogue of Brent's lemma [17] for parallel and hierarchical computations.

Theorem 10 *Consider a program $\mathcal{P}$ of a D-BSP$(v, \mu, g(x))$, where each processor performs local computation for $O(\tau)$ time, and there are λ_i i-supersteps, for $0 \leq i < \log v$. If $g(x)$ is $(2, c)$-uniform, then for any $1 \leq v' \leq v$, $\mathcal{P}$ can be simulated in time*

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

on D-BSP$(v', \mu v/v', g(x))$.

PROOF. Let us refer to D-BSP$(v, \mu, g(x))$ and the D-BSP$(v', \mu v/v', g(x))$ 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 local memory of P_j is organized into v/v' blocks of μ cells each, with block i containing the context of the i-th processor of $C_j^{(\log v')}$.

The simulation strategy deals differently with supersteps of label $i < \log v'$ and $i \geq \log v'$. Specifically, the simulation regards program $\mathcal{P}$ as partitioned into maximal runs of consecutive supersteps whose labels are either all less than $\log v'$ or all greater than or equal to $\log v'$. Consider a run of the first type. The supersteps of the run are simulated one after the other: in particular, each i-superstep of the guest machine is simulated through an i-superstep followed by a $\log v'$-superstep on the host machine. In the i-superstep, each host processor P_j takes care of the local computation of its assigned guest processors by iteratively moving their contexts to the top of memory and then sending the messages generated by the v/v' guest processors to the host processors associated with the message destinations. In the subsequent $\log v'$-superstep, each host processor moves the received messages to the incoming message buffers of the appropriate guest processors.

A run of supersteps with labels greater than or equal to $\log v'$ is simulated in parallel on each host processor as if it were a program $\mathcal{P}'$ designed for a D-BSP$(v/v', \mu, g(x))$, regarding an i-superstep of $\mathcal{P}$ as an $(i - \log v')$-superstep in $\mathcal{P}'$. This simulation is accomplished by a straightforward adaptation of the strategy presented in Section 3 (the only difference being that $\mathcal{P}$ may not be fine-grained).

Let us now evaluate the running time of the simulation. An i-superstep in a run of the first type, requiring $O(\tau')$ local computation time at each guest processor and the execution of an h-relation, is simulated in time

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

where the term $(v/v')hg(\mu v/2^i)$ accounts for the execution of an $h(v/v')$-relation within i-clusters on the host, while the term $(v/v')\mu g(\mu v/v')$ accounts for the cost of moving the guest processors' contexts to the top of memory during the simulation of the v/v' local computations. Since $h \leq \mu$ and $i < \log v'$, the above formula simplifies to

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

Consider now a run of supersteps of labels greater than or equal to $\log v'$. Suppose that the run comprises δ_i i-supersteps, with $\log v' \leq i \leq \log v$, and that each guest processor performs $O(\tau')$ local computation in the run. By applying Theorem 5, the run is simulated in time

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

The theorem follows by adding up the contributions of all the runs.

We call a program for a D-BSP$(v, \mu, g(x))$ *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. Obviously, a fine-grained program is full. The following corollary establishes the analogue of Brent's lemma for full programs.

Corollary 11 *If $g(x)$ is $(2, c)$-uniform, then any T-time full program for a D-BSP$(v, \mu, g(x))$ can be simulated in time $\Theta(Tv/v')$ on a D-BSP$(v', \mu v/v', g(x))$, for any $1 \leq v' \leq v$.*

5 Simulation of D-BSP on BT

In this section we present an algorithm that simulates a fine-grained D-BSP$(v, \mu, g(x))$ 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 $0 < \alpha < 1$, 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 [2] are captured by the above scenario.

We adopt the same overall simulation strategy as the one presented in Section 3. However, in order to exploit spatial locality effectively, as encouraged by the BT model, the actual implementation of the strategy requires crucial modifications, which are described in Subsections 5.1 and 5.2. The application of the simulation to relevant problems is discussed in Subsection 5.3.

5.1 Memory Layout

Although the simulation algorithm of Section 3 yields a valid BT program, it is not designed to exploit block transfer. For example, in Step 2 the algorithm brings one context at a time to the top of memory and simulates communications touching the contexts in a random fashion, which is highly inefficient in the BT framework. As suggested in [2], a good BT algorithm should be recursive, with block transfer used at every level of recursion. Since the BT model supports block copy operations only for non-overlapping memory regions, additional buffer space is required to perform swaps of large chunks of data; moreover, in order to minimize access costs, such buffer space must be allocated close to the blocks to be swapped. As a consequence, the required buffers must be interspersed with the contexts.

During the simulation, buffer space is dynamically created or destroyed by means of the 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:

if $i = \log v$ **then return**
Copy blocks $v/2^{i+1}, \ldots, v/2^i - 1$ onto blocks $v/2^i, \ldots, 3v/2^{i+1} - 1$
UNPACK($i + 1$)

Note that the copy operation executed by UNPACK can be performed with a single block transfer. 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 follow C among the contexts of C itself. Figure 4 illustrates how the memory layout is modified by a call to UNPACK(0) when $v = 8$. It is not difficult to prove that the buffer creation 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 $(2, c)$-uniform, we can conclude that the buffers do not alter memory access time by more than a multiplicative constant.

Subroutine PACK(i) performs the same operations of UNPACK(i) but in reverse order, thus compacting the contexts belonging to the (unpacked) topmost i-cluster. (The code is omitted for brevity.) A simple recurrence proves that the complexity of UNPACK(i) is dominated by the initial movement of $v/2^{i+1}$ blocks, which takes time proportional to the size of the i-cluster.

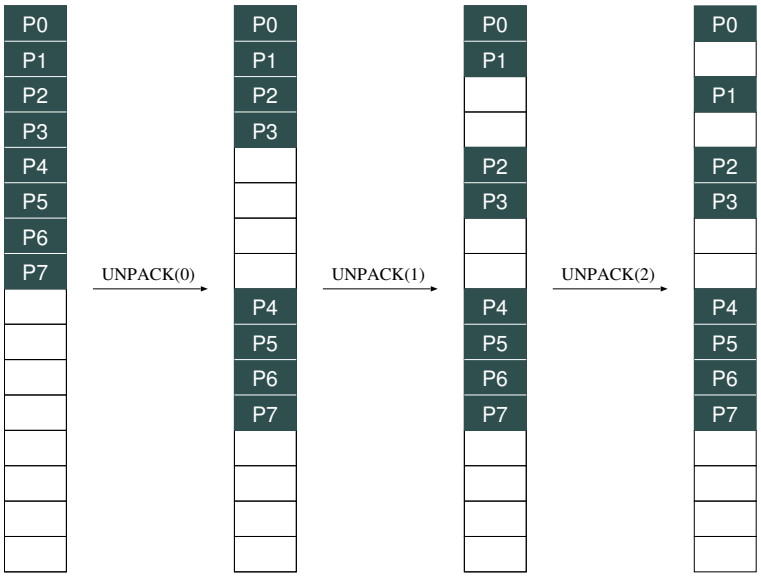

Fig. 4. Snapshots of the BT memory layout during an UNPACK(0) operation. The host D-BSP machine has 8 processors. Solid boxes indicate processor contexts, and white boxes indicate empty buffers.

Clearly, the same complexity applies to PACK(i). Thus, the running times of the two subroutines are

$$T_{\text{UNPACK}}(i), T_{\text{PACK}}(i) = O\left(\mu v/2^i\right).$$

5.2 The simulation algorithm

The structure of the simulation algorithm, whose pseudocode is given in Figure 5, is identical to the one of Section 3, except that suitable calls to UNPACK and PACK have been introduced in order to create the required buffer space needed to perform swaps that efficiently exploit block transfer.

Observe that at the beginning of each round, the overall memory layout is the same as the one resulting from a call to UNPACK(0). Moreover, the augmented code still maintains Invariants 1 and 2 stated in Section 3 except that, in Invariant 2, the contexts of each cluster C maintain the same relative order but are now interspersed with at most $|C|$ empty blocks. As a consequence, the correctness of the simulation can be proved using the same argument of Theorem 4 if we assume that the input program is $\mathcal{L}$-smooth, for some set of labels $\mathcal{L}$. If the program $\mathcal{P}$ to be simulated is not $\mathcal{L}$-smooth, it is first transformed into a functionally equivalent $\mathcal{L}$-smooth program $\mathcal{P}'$, as discussed in Section 3. Note that the correctness does not depend on the specific set $\mathcal{L}$

```
0     UNPACK(0)
      while true do
1         P ← processor whose context is on top of memory
          s ← superstep number to be simulated next for P
          C ← i_s-cluster containing P
1.a       PACK(i_s)
2         Simulate Superstep s for C
3         if P has finished its program then exit
4         if i_{s+1} < i_s then
              b ← 2^{i_s − i_{s+1}}
              Let Ĉ be the i_{s+1}-cluster containing C,
                  and let Ĉ_0 ... Ĉ_{b−1} be its component i_s-clusters,
                  with C = Ĉ_j for some index j
              if j > 0 then swap the contexts of C with those of Ĉ_0
              if j < b − 1 then
                  swap the contexts of Ĉ_0 with those of Ĉ_{j+1}
5         UNPACK(i_s)
```

Fig. 5. The revised simulation algorithm

with respect to which the program is made smooth. However, the choice of $\mathcal{L}$ may affect the simulation time and we will postpone it to Subsection 5.2.2.

5.2.1 Implementation of Step 2

As mentioned before, in order to exploit block transfer, the simulation of a Superstep s for an i_s-cluster C performed in Step 2 must be carefully restructured. Specifically, we organize it in two phases: first, local computations are executed in a recursive fashion, and then the communications required by the superstep are simulated.

Simulation of local computations. By virtue of the aforementioned invariant and by Step 1.a, the simulation of Superstep s for cluster C begins with all contexts of C packed at the top of memory, followed by $|C|$ empty blocks. In order to exploit both temporal and spatial locality, processor contexts are iteratively brought to the top of memory in chunks of suitable size, and the prescribed local computation is then performed for each chunk recursively. Let n be a power of 2 and define $c(n)$ as the greatest power of 2 such that $c(n) \leq \min\{f(\mu n)/\mu, n/2\}$. The local computation of Superstep s for C is simulated by invoking the recursive function COMPUTE(n), with $n = v/2^{i_s}$, whose code is given in Figure 6.

The correctness of COMPUTE(n) can be proved by induction, based on

```
COMPUTE(n)
    if n = 1 then
        Simulate local computation for the context in block 0
    else
        c ← c(n)
        t ← n/c {number of chunks}
        Shift blocks c, ..., n − 1 to blocks 2c, ..., n + c − 1
        COMPUTE(c)
        for j ← 2 to t do
            Swap blocks 0, ..., c − 1 with blocks jc, ..., (j + 1)c − 1
            COMPUTE(c)
            Swap blocks jc, ..., (j + 1)c − 1 with blocks 0, ..., c − 1
        Shift blocks 2c, ..., n + c − 1 onto blocks c, ..., n − 1
```

Fig. 6. The COMPUTE subroutine.

the observation that each of the recursive calls to COMPUTE(c) starts and ends with the c processor contexts packed on top of memory, followed by c empty blocks, which leaves sufficient space to perform the swaps using block transfer.

Since local computation for a processor is always performed while the corresponding context is stored in the topmost memory block, the running time of COMPUTE(n) is given by the sum of the original computation times for the guest processors belonging to cluster C, plus the overhead caused by the memory movements. Let $T_{\mathrm{M}}(n)$ denote this overhead. Since each shift or swap operation requires a constant number of block transfers, it is easy to see that

$$T_{\mathrm{M}}(n) = (n/c(n))T_{\mathrm{M}}(c(n)) + O\left(\mu n + \sum_{j=1}^{n/c(n)} f(\mu j c(n))\right).$$

By noting that $\sum_{j=1}^{n/c(n)} f(\mu j c(n)) = O\left((n/c(n))f(\mu n)\right)$ and by applying the definition of $c(n)$, we conclude that the additive term in the equation for the case $n > 1$ is $O(\mu n)$. Let $c^*(x) = \min\{k \geq 1 : c^{(k)}(x) \leq 1\}$. Then, it can be seen that

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

As specific instances, we have that $T_{\mathrm{M}}(n)$ is $O(\mu n \log^*(\mu n))$ if $f(x) = \log x$, and $O(\mu n \log\log(\mu n))$ if $f(x) = x^\alpha$. For the purposes of our simulation it is sufficient to say that $T_{\mathrm{M}}(n) = O(\mu n \log\log(\mu n))$ for any $f(x) = O(x^\alpha)$. Therefore, we have that the time required to simulate the local computation

of i_s-cluster C for Superstep s is

$$O\left((v/2^{i_s})\tau_s + (\mu v/2^{i_s})\log\log(\mu v/2^{i_s})\right), \tag{2}$$

where τ_s is the maximum computation time spent by any processor in C during Superstep s.

Simulation of communications. The second phase of the simulation of Superstep s for i_s-cluster C takes care of the communication prescribed by the superstep. For ease of presentation, we suppose that the incoming and outgoing message buffers of each processor are located at the end of its context. To deliver all messages to their destinations, we make use of sorting. Specifically, the contexts of C are divided into $\Theta(\mu|C|)$ constant-sized elements, which are then sorted in such a way that after the sorting, contexts are still ordered by processor number and all messages destined to processor P_j are stored at the end of P_j's context. This is easily achieved by sorting elements according to suitably chosen tags attached to the elements, which can be produced during the simulation of local computation without asymptotically increasing the running time.

Although the idea behind the message delivery phase is quite simple, there are two technical issues that must be dealt with. First of all, the sorting algorithm must use block transfer effectively. To satisfy this requirement, we employ the *Approx-Median-Sort* algorithm proposed in [2]. This algorithm is capable of sorting m constant-sized items in time $O(m\log m)$ if $f(x) = O(x^\alpha)$, for any constant $0 < \alpha < 1$; unfortunately, the algorithm requires $\Theta(m\log\log m)$ space. In our case, the number of elements to sort is $\Theta(\mu v/2^{i_s})$ so the required memory space is

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

Our buffer policy ensures that when we start simulating the message exchange, the i_s-cluster is followed by an empty space of size $\mu v/2^{i_s}$, which is clearly not enough for sorting. 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. Let $i_k < i_s$ be the biggest integer such that $\mu v/2^{i_k} \ge L(i_s)$, or 0 if $\mu v < L(i_s)$.

Then, we can free a sufficient amount of space for sorting through the following steps.

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

It is easily seen that these steps can be completed in $O\left(L(i_s)\right)$ time. Clearly, after sorting is completed, the same steps must be executed in reverse order.

The second technical issue arises since the message delivery phase may alter the size (hence the position) of the contexts. Indeed, the size of a processor's context after sorting depends on the amount of data received by the processor. As a consequence, it is necessary to realign the contexts so that the context of the j-th processor of C ends up again in memory block j, for $0 \leq j < |C|$. This operation can be performed by the following recursive subroutine (initially invoked with $n = |C| = v/2^{i_s}$).

ALIGN(n)
    if $n = 1$ **then exit**
    Locate the $(n/2)$-th topmost context
    Copy contexts $n/2, \ldots, n-1$ to the memory region
        that starts with block n
    ALIGN($n/2$)
    Swap blocks $0, \ldots, n/2 - 1$ with blocks $n, \ldots, 3n/2 - 1$
    ALIGN($n/2$)
    Copy blocks $0, \ldots, n/2 - 1$ onto blocks $n/2, \ldots, n-1$
    Copy blocks $n, \ldots, 3n/2 - 1$ onto blocks $0, \ldots, n/2 - 1$

A context can be easily located through binary search over the tags. It is easy to see that each recursive call to ALIGN(x) is made with x contexts occupying (at most) the top x blocks, followed by x empty blocks. This provides sufficient space for the swaps to be performed. A simple analysis shows that the running time of ALIGN(n) is $O\left(\mu n \log \mu n\right)$, which is the same time taken by sorting.

Since the time required by the creation of buffer space prior to sorting and the corresponding recompaction at the end is dominated by the sorting time, we conclude that the simulation of the message exchange prescribed by Superstep s for i_s-cluster C can be accomplished in time

$$O\left((\mu v/2^{i_s}) \log(\mu v/2^{i_s})\right). \tag{3}$$

To conclude this section, Figure 7 summarizes the steps which are necessary to simulate Superstep s for cluster C.

COMPUTE($v/2^{i_s}$)
$i_k \leftarrow$ biggest integer such that $\mu v/2^{i_k} \geq L(i_s)$
UNPACK(i_s)
PACK(i_k)
Shift blocks $v/2^{i_s}, \ldots, v/2^{i_k} - 1$ to the memory region
    that starts with block $v/2^{i_s} + \lceil L(i_s)/\mu \rceil$
Divide the contexts of C into constant sized-elements
    and sort them via Approx-Median-Sort
ALIGN($v/2^{i_s}$)
Shift blocks $v/2^{i_s}, \ldots, v/2^{i_k} - 1$ back
    to their original positions
UNPACK(i_k)
PACK(i_s)

Fig. 7. Pseudocode for Step 2 in the revised simulation algorithm.

5.2.2 Analysis of the simulation time

In order to analyze the running time of the simulation algorithm on $f(x)$-BT, we need to define a suitable set $\mathcal{L}$ of superstep labels with respect to which the input program $\mathcal{P}$ is transformed into an equivalent $\mathcal{L}$-smooth program $\mathcal{P}'$. Labels in $\mathcal{L}$ must be carefully chosen so that the smoothing process does not unduly increase the asymptotic complexity of the simulation, which will be eventually expressed in terms of the original program $\mathcal{P}$, rather than $\mathcal{P}'$. To this purpose, we set $\mathcal{L} = \{0 = \ell_0 < \ell_1 < \ldots < \ell_m = \log v\}$ such that, for suitable constants c_1, c_2, d_1, d_2, with $0 < c_1 \leq c_2 < 1$ and $d_1, d_2 > 1$, it holds that: (a) $\log(d_1 \mu v/2^{\ell_{i+1}}) \geq c_1 \log(d_1 \mu v/2^{\ell_i})$, for every $0 \leq i < m$; (b) $\log(d_1 \mu v/2^{\ell_{i+1}}) \leq c_2 \log(d_1 \mu v/2^{\ell_i})$, for every $0 \leq i < m-1$; and (c) $f(\mu v/2^{\ell_i}) \leq d_2 \mu v/2^{\ell_{i+1}}$, for every $0 \leq i < m$. It can be shown that such a set $\mathcal{L}$ is well defined by means of techniques similar to the ones employed in the analysis of the HMM simulation. More specifically, recalling that $f(x) = O(x^\alpha)$ with $\alpha < 1$, fix an arbitrary value c_2, with $\alpha < c_2 < 1$ and fix d_1 so that $\log(d_1 x)$ is $(2, c)$-uniform for $c = c_2/\alpha$. Then $\mathcal{L}$ can be determined as before and the three properties follow with $c_1 = c_2/c = \alpha$ and $d_2 = d_1^{1-\alpha}$.

First, consider Step 2. By combining Equations 2 and 3, we conclude that the simulation of Superstep s for i_s-cluster C requires time $O((v/2^{i_s})\tau_s + (\mu v/2^{i_s}) \log(\mu v/2^{i_s}))$. Therefore, the overall contribution of Su-

perstep s to the simulation time is

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

Observe that the above running time dominates the aggregate time required by the PACK and UNPACK operations (Steps 1.a and 5) performed in the rounds simulating the superstep. Note also that Step 2 needs not be executed for the dummy supersteps introduced in the smoothing process. Therefore, the contribution of each such superstep to the simulation time is only due to the PACK and UNPACK operations, which is $O(\mu v)$.

When $i_{s+1} < i_s$, the simulation algorithm incurs the additional cost due to the need of moving the $2^{i_s - i_{s+1}}$ sibling i_s-clusters internal to an i_{s+1}-cluster to the top of memory, as prescribed by Step 4 of the pseudocode presented in Figure 5. Thanks to the availability of buffer space, the swaps required by the j-th such move, $0 \leq j < 2^{i_s - i_{s+1}}$, can be performed with at most three block transfers taking time $O(f(\mu j|C|) + \mu|C|)$. By summing over all values of j, we have that the overhead incurred by cluster swaps before the simulation of an i_{s+1}-cluster is

$$O\left(2^{i_s - i_{s+1}} f(\mu v/2^{i_{s+1}}) + \mu v/2^{i_{s+1}}\right) = O\left(\mu v/2^{i_{s+1}}\right),$$

where the equality follows by the $\mathcal{L}$-smoothness of the program. Therefore, the aggregate cost of all executions of Step 4 in the rounds simulating Superstep s is amortized by the cost of the future simulation of Superstep $s+1$, hence it can be ignored.

In conclusion, the time for simulating $\mathcal{P}'$ is simply the sum of the times for the simulation of its supersteps, namely

$$O\left(v\left(\tau + \mu \sum_{\ell \in \mathcal{L}} \lambda'_\ell \log(\mu v/2^\ell)\right)\right),$$

where τ is the maximum local computation time of any processor and λ'_ℓ denotes the number of ℓ-supersteps executed by $\mathcal{P}'$, for $\ell \in \mathcal{L}$. In order to obtain an expression for the simulation time dependent on the parameters of $\mathcal{P}$, rather than those of $\mathcal{P}'$, we observe that each i-superstep of $\mathcal{P}$ is upgraded in $\mathcal{P}'$ to an ℓ-superstep, where i and ℓ are such that $\log(\mu v/2^i) = \Theta\left(\log(\mu v/2^\ell)\right)$. Moreover, since $f(x) = O(x^\alpha)$, it is easy to see that in $\mathcal{P}'$ there cannot be more than $O(\log\log(\mu v/2^j))$ dummy supersteps introduced between a pair of

consecutive original supersteps of labels i and j, with $j < i$. Therefore, the overall simulation cost of each such run of dummy supersteps is amortized by the cost of simulating the original superstep closing the run.

The above discussion establishes the following result.

Theorem 12 *Consider a fine-grained D-BSP$(v, \mu, g(x))$ program $\mathcal{P}$, where each processor performs local computation for $O(\tau)$ time, and there are λ_i i-supersteps for $0 \leq i \leq \log v$. For any $(2, c)$-uniform access function $f(x) = O(x^\alpha)$, with constant $0 < \alpha < 1$, $\mathcal{P}$ can be simulated on $f(x)$-BT in time $O\left(v\left(\tau + \mu \sum_{i=0}^{\log v} \lambda_i \log(\mu v/2^i)\right)\right)$.*

We remark that, besides the unavoidable term $v\tau$, the complexity of the sorting operations performed in Step 2 is the dominant factor in the simulation time. Moreover, it is important to observe that, unlike the HMM case, the simulation time in the above theorem *does not* depend on $f(x)$. This is in accordance with the findings of [2], which show that an efficient exploitation of the powerful block transfer capability of the BT model is able to hide access costs almost completely.

5.3 Application to Case-Study Problems

In this subsection we provide evidence of the effectiveness of our simulation by showing how it can be employed to obtain efficient BT algorithms starting from D-BSP ones. For the sake of comparison, we observe that Fact 2 implies that for relevant access functions $f(x)$, any straightforward approach simulating one entire superstep after the other would require time $\omega(v)$ per superstep just for touching the v processor contexts, while our algorithm can overcome such a barrier by carefully exploiting submachine locality.

First consider the n-MM problem. It is easily seen that the n-processor D-BSP algorithm described in Subsection 3.1 uses 2^i $2i$-supersteps, for every $0 \leq i < \log(n)/2$, performing constant local computation per superstep. By applying Theorem 12 our simulation of this algorithm yields an optimal $O\left(n^{3/2}\right)$ algorithm for $f(x)$-BT, while a trivial step-by-step simulation would have required at least time $\Omega\left(n^{3/2} \log^* n\right)$ for $f(x) = \log x$, and time $\Omega\left(n^{3/2} \log\log n\right)$ for $f(x) = x^\alpha$.

In general, different D-BSP bandwidth functions $g(x)$ may promote different algorithmic strategies for the solution of a given problem. Therefore,

without a strict correspondence between $g(x)$ and the BT access function $f(x)$ in the simulation, the question arises of which function $g(x)$ suggests the best "coding practices" for BT. Unlike the HMM scenario (see Corollary 6), the choice $g(x) = f(x)$ is not always the best. Consider, for instance, the n-DFT problem. As mentioned in the proof of Proposition 8, two D-BSP algorithms for this problem are applicable. The first algorithm is a standard execution of the n-input FFT dag and requires one i-superstep, for $0 \leq i < \log n$. The second algorithm is based on a recursive decomposition of the same dag into two layers of $\sqrt{n}$ independent $\sqrt{n}$-input subdags, and can be shown to require 2^i supersteps with label $(1 - 1/2^i) \log n$, for $0 \leq i < \log \log n$. On D-BSP$(n, O(1), x^\alpha)$, both algorithms yield a running time of $O(n^\alpha)$, which is clearly optimal. However, the simulations of these two algorithms on the x^α-BT take time $O\left(n \log^2 n\right)$ and $O(n \log n \log \log n)$, respectively. This implies that the choice $g(x) = f(x)$ is not *effective* [23], in the sense that D-BSP$(n, O(1), x^\alpha)$ does not reward the use of the second algorithm over the first one. On the other hand, D-BSP$(n, O(1), \log x)$ correctly distinguishes among the two algorithms, since their respective parallel running times are $O\left(\log^2 n\right)$ and $O(\log n \log \log n)$.

The above example is a special case of the following more general consideration. Observe that Theorem 12 yields a simulation with linear slowdown for any D-BSP$(v, O(1), \log x)$ program on $f(x)$-BT, and that a program for D-BSP$(v, O(1), g(x))$ is also valid for D-BSP$(v, O(1), \log x)$. Combining these observations, it can be argued that the choice $g(x) = \log x$ yields the most effective instance of the D-BSP model for obtaining sequential algorithms for the class of $f(x)$-BT machines. Indeed, observe that given two D-BSP algorithms A_1, A_2 solving the same problem, if the simulation of A_1 on $f(x)$-BT runs faster than the simulation of A_2, then A_1 exhibits a better asymptotic performance than A_2 also on D-BSP$(v, O(1), \log x)$.

6 Conclusions

The fact that code written with parallelism in mind can run efficiently on platforms with hierarchical memory, is a widespread belief in the computer science community. Prior results in the literature have mostly related flat parallelism (i.e., parallelism in machine models with symmetric communication infrastructures) to efficiency on two-level hierarchies. In this paper we have

embarked on the investigation of a more general scenario by considering the relation between locality of communications in parallel programs (as exposed by submachine locality) and locality of reference (both temporal and spatial) in multi-level hierarchies.

In particular, we have developed efficient schemes to simulate parallel programs written for the D-BSP, a model where submachine locality can be explicitly exposed, on the sequential HMM model, which rewards the exploitation of temporal locality, and on its extension with block transfer, the BT model, which also rewards the exploitation of spatial locality.

The simulation on the HMM achieves linear slowdown when the bandwidth function $g(x)$ of D-BSP coincides with the HMM access function $f(x)$, thus charging communications in a D-BSP cluster as accesses to a hierarchical memory with size proportional to that of the cluster's aggregate memory. This result provides evidence of a strict relation between submachine locality and temporal locality. Further evidence comes from the observation that efficient algorithms designed on the D-BSP for prominent problems are translated by our simulation into optimal HMM algorithms. Moreover, by regarding each individual D-BSP processor as an HMM, the simulation can be extended to provide an analogue of Brent's lemma for parallel and hierarchical platforms. In this respect, the augmented D-BSP model features a seamless integration of parallelism and memory hierarchy.

Unlike the HMM case, the running time of the D-BSP simulation on $f(x)$-BT is independent of the access function $f(x)$. Intuitively, this phenomenon can be attributed to the exploitation of spatial locality afforded by block transfer, which flattens, in part, the access costs to the memory hierarchy. This finding is in accordance with the results of [2], which show how relevant problems (such as sorting, FFT, matrix multiplication) require the same time on such diverse machines as x^α-BT and $\log(x)$-BT. Nevertheless, our simulation is still able to transform optimal D-BSP algorithms into optimal (or quasi optimal) BT ones. Moreover, linear slowdown can be achieved when simulating D-BSP$(v, O(1), \log x)$ on the $f(x)$-BT, for any $f(x) = O(x^\alpha)$, which makes D-BSP$(v, O(1), \log x)$ the most effective choice for obtaining efficient BT algorithms from parallel ones.

Finally, we remark that the proposed simulation cannot be further improved in the general case, since the lower bound proved in [2] on the execution of random permutations on $f(x)$-BT can be employed to design a D-BSP pro-

gram for which the time of our simulation algorithm is the least possible. However, an improved simulation can be obtained when the communication patterns generated by the algorithm are known *a priori* and exhibit certain regularities. As an example, consider again the $O(\log n \log\log n)$-time algorithm for the n-DFT designed for D-BSP$(n, O(1), \log x)$. By simulating the transpose permutation generated by each superstep of the algorithm by the rational permutation algorithm in [2], 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-optimality is 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).

Acknowledgements

The authors are grateful to one of the anonymous reviewers for a number of profound suggestions that helped improve the quality of the paper. They also wish to thank Gianfranco Bilardi and Franco Preparata for fruitful discussions and insights on the subject of this work.

References

[1] A. Aggarwal, B. Alpern, A. Chandra, M. Snir, A model for hierarchical memory, in: Proc. of the 19th ACM Symp. on Theory of Computing, 1987, pp. 305–314.

[2] A. Aggarwal, A. Chandra, M. Snir, Hierarchical memory with block transfer, in: Proc. of the 28th IEEE Symp. on Foundations of Computer Science, 1987, pp. 204–216.

[3] J. Vitter, E. Shriver, Algorithms for parallel memory II: Hierarchical multilevel memories, Algorithmica 12 (2/3) (1994) 148–169.

[4] A. Aggarwal, J. Vitter, The input/output complexity of sorting and related problems, Communications of the ACM 31 (9) (1988) 1116–1127.

[5] J. Vitter, E. Shriver, Algorithms for parallel memory I: Two-level memories, Algorithmica 12 (2/3) (1994) 110–147.

[6] B. Alpern, L. Carter, E. Feig, T. Selker, The uniform memory hierarchy model of computation, Algorithmica 12 (2/3) (1994) 72–109.

[7] S. Sen, S. Chatterjee, N. Dumir, Towards a theory of cache-efficient algorithms, Journal of the ACM 49 (6) (2002) 828–858.

[8] F. Dehne, W. Dittrich, D. Hutchinson, Efficient external memory algorithms by simulating coarse-grained parallel algorithms, Algorithmica 36 (2) (2003) 97–122.

[9] F. Dehne, D. Hutchinson, D. Maheshwari, W. Dittrich, Bulk synchronous parallel algorithms for the external memory model, Theory of Computing Systems 35 (6) (2002) 567–597.

[10] J. Sibeyn, M. Kaufmann, BSP-like external-memory computation, in: Proc. of 3rd CIAC, LNCS 1203, 1999, pp. 229–240.

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

[12] A. Bäumker, W. Dittrich, F. Meyer auf der Heide, Truly efficient parallel algorithms: 1-optimal multisearch for and extension of the BSP model, Theoretical Computer Science 203 (1998) 175–203.

[13] F. Dehne, A. Fabri, A. Rau-Chaplin, Scalable parallel geometric algorithms for coarse grained multicomputers, International Journal on Computational Geometry 6 (3) (1996) 379–400.

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

[15] Y. Chiang, M. Goodrich, E. Grove, R. Tamassia, D. Vengroff, J. Vitter, External-memory graph algorithms, in: Proc. of the 6th ACM-SIAM Symp. On Discrete Algorithms, 1995, pp. 139–149.

[16] G. Bilardi, F. Preparata, Processor-time tradeoffs under bounded-speed message propagation: Part I, upper bounds, Theory of Computing Systems 30 (1997) 523–546.

[17] R. Brent, The parallel evaluation of general arithmetic expressions, Journal of the ACM 21 (2) (1974) 201–208.

[18] G. Bilardi, F. Preparata, Processor-time tradeoffs under bounded-speed message propagation: Part II, lower bounds, Theory of Computing Systems 32 (1999) 531–559.

[19] P. De la Torre, C. Kruskal, Submachine locality in the bulk synchronous setting, in: Proc. of EUROPAR 96, LNCS 1124, 1996, pp. 352–358.

[20] G. Bilardi, E. Peserico, A characterization of temporal locality and its portability across memory hierarchies, in: Proc. of 28th Int. Colloquium on Automata, Languages and Programming, LNCS 2076, 2001, pp. 128–139.

[21] G. Bilardi, K. Ekanadham, P. Pattnaik, Optimal organizations for pipelined hierarchical memories, in: Proc. of the 14th ACM Symp. on Parallel Algorithms and Architectures, 2002, pp. 109–116.

[22] G. Bilardi, C. Fantozzi, A. Pietracaprina, G. Pucci, On the effectiveness of D-BSP as a bridging model of parallel computation, in: Proc. of the Int. Conference on Computational Science, LNCS 2074, 2001, pp. 579–588.

[23] G. Bilardi, A. Pietracaprina, G. Pucci, A quantitative measure of portability with application to bandwidth-latency models for parallel computing, in: Proc. of EUROPAR 99, LNCS 1685, 1999, pp. 543–551.

[24] C. Fantozzi, A. Pietracaprina, G. Pucci, A general PRAM simulation scheme for clustered machines, Intl. Journal of Foundations of Computer Science 14 (6) (2003) 1147–1164.

Carlo Fantozzi obtained the Laurea (2000) in Computer Engineering (*summa cum laude*) from the University of Padova, Italy, where he also received the PhD (2004) in Computer Engineering. He is now a designer of embedded systems at CAREL SpA, Italy, where he is a member of the Technical Department. His research interests encompass algorithms and architectures for high performance computing, both at the processor and at the system scale levels, and the management of distributed systems. He has authored several papers that appeared in international conferences and journals.

Andrea Pietracaprina received the Laurea (1987) in Computer Science (*summa cum laude*) from the University of Pisa, Italy and the MS (1991) and PhD (1994) both in Computer Science from the University of Illinois at Urbana-Champaign, USA. In 1994, he spent a year as a Postdoctoral Fellow at the University of Padova. From December 1994 to November 1998 he was an Assistant Professor at the Department of Pure and Applied Mathematics of the same university. Since 1998, he has been with the Department of Information Engineering of the University of Padova, where he is currently a Professor of Computer Science. Prof. Pietracaprina's main research interests lie in the fields of high-performance computation, data mining, network routing, and combinatorial optimization. He is the author of over 40 papers in international journals and conferences. Prof. Pietracaprina is a member of ACM and IEEE.

Geppino Pucci received the Laurea (1987) (*summa cum laude*) and the PhD (1993) degrees both in Computer Science from the University of Pisa, Italy. From 1988 to 1990 he was with the Computing Laboratory of the University of Newcastle-upon-Tyne, United Kingdom, as a research associate. He spent 1993 at the International Computer Science Institute, Berkeley, California, as a postdoctoral fellow. Since 1992, he has been with the Department of Information Engineering of the University of Padova, Italy, where he is currently a Professor of Computer Science. His research interests include design and analysis of parallel algorithms, theory of computation, probabilistic model-

ing, and algorithm engineering for problems in computational sciences. On these subjects, he has authored about 50 papers in international journals and conferences. Prof. Pucci is a member of ACM and IEEE.