# Translating Submachine Locality into Locality of Reference (Extended Abstract)*

Carlo Fantozzi Andrea Pietracaprina Geppino Pucci
Dept. of Information Engineering, University of Padova, Padova Italy
{carlo.fantozzi,andrea.pietracaprina,geppino.pucci}@dei.unipd.it

## Abstract

*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. Recently, 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 work, we complete and extend the above result by encompassing also spatial locality. Specifically, we present a scheme to simulate parallel algorithms designed for the Decomposable BSP (a BSP variant which captures submachine locality) on the Hierarchical Memory Model with Block Transfer. The simulation yields good hierarchy-conscious sequential algorithms from parallel ones, and provides evidence of the strict relation between submachine locality in parallel computation and locality of reference (both temporal and spatial) in the hierarchical memory setting.*

## 1. Introduction

Modern memory systems feature a complex multilevel hierarchical structure, with capacity and access times increasing as levels grow farther from the CPU. In order to amortize the larger costs incurred when accessing distant levels of the hierarchy, data is transferred between the levels in blocks of contiguous locations. The main idea behind the efficient exploitation of such a hierarchical organization is that an algorithm will be able to reduce memory access costs by organizing the computation so that the same data are frequently reused within a short time interval, and that consecutive data in memory are involved in consecutive operations, two properties known as *temporal* and *spatial locality of reference*, respectively.

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 computational models have been proposed which 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 $f(x)$, thus encouraging the exploitation of temporal locality. The *Hierarchical Memory Model with Block Transfer* model (BT, for short) [2, 21] 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 [3, 20], which has been extensively used in the literature to develop I/O-efficient algorithms. Finally, the cache models defined in [15, 16] have been at the base of the vast literature on cache-efficient algorithms.

Earlier works provided evidence that efficient sequential algorithms for two-level hierarchies can be obtained by simulating parallel ones. In [10, 12, 17] schemes are presented that simulate parallel algorithms designed for coarse-grained parallel models, such as BSP [18], BSP* [4], CGM [11], 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 [8] based on the simulation of fine-grained PRAM algorithms. Beside proving a gen-

*This research was supported in part by MIUR of Italy under project "ALINWEB: Algorithmics for Internet and the Web", and by the University of Padova under Grant CPDA033838.

eral 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 [19] where parallelism is turned into efficient cache prefetching strategies.

A more general study on the relation between parallelism and locality of reference has been initiated in [13], where we have shown how a more structured form of parallelism, such as the one exhibited by the Decomposable BSP (a clustered variant of BSP defined in [9]) can lead to the design of sequential algorithms that feature an optimal exploitation of temporal locality on arbitrarily deep memory hierarchies. More precisely, the main contribution of that work is a strategy that simulates a D-BSP algorithm on HMM with a slowdown which is merely proportional to the loss of parallelism, thus proving that no extra cost is incurred in accessing the much larger memory of the sequential machine. This result crucially relies on a nontrivial exploitation of the submachine locality exposed in D-BSP to organize the sequential computation into phases where accesses are confined to small subsets of data.

The objective of the current paper is to continue the above investigation to encompass 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, building on the results of [13] we devise an efficient strategy to simulate D-BSP algorithms on the BT model. The simulation 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 in [13], 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 results 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 BT 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 describes and analyzes the simulation algorithm. Finally, section 4 discusses several implications of our result and its application to some relevant case studies.

## 2. Machine Models

**BT.** The $f(x)$-BT (Hierarchical Memory Model with Block Transfer) was introduced in [2] by augmenting the $f(x)$-HMM model of [1] with a block transfer facility. Specifically, as in the HMM, an access to memory location $x$ requires time $f(x)$, for a given nondecreasing function $f(x)$, 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 all other works in the literature, we will focus our attention on *polynomially bounded* access functions, that is, functions $f(x)$ for which there exists a constant $c$ such that $f(2x) \leq cf(x)$, for any $x$. Particularly interesting and widely studied special cases are the functions $f(x) = x^\alpha$, for a positive constant $\alpha < 1$, and $f(x) = \log x$.

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. The following proposition is proved in [2].

**Proposition 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 [5].

**D-BSP.** The *Decomposable Bulk Synchronous Parallel* (D-BSP) model was introduced in [9] to capture submachine locality in a structured way through submachine decomposition, and was further investigated in [6, 7, 14].

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 $\mu$. 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 (an output and an input queue for message exchange are part of each 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. If each processor spends at most $\tau$ 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 $\tau + hg(\mu v/2^i)$. With this particular choice of the cost function, communication within an $i$-cluster is envisioned as a sort of remote access with access function $g(x)$ outside the aggregate memory of the cluster.

Since the objective of this paper is to assess to what extent submachine locality can be transformed into locality of reference, we will concentrate on 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.

## 3. The Simulation Algorithm

In this section we present an algorithm that simulates a 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. Moreover, as mentioned before, we concentrate on the case $\mu = O(1)$.

We adopt the same overall simulation strategy as the one presented in [13] for the HMM model, which we briefly recall below. The memory of the host machine is divided into *blocks* of $\mu$ 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. The simulation is organized into 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 for the execution of the next round. Specifically, 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). 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 algorithm in figure 1 iteratively simulates ready clusters whose processor contexts are located consecutively in the topmost memory blocks.

**while** true **do**
1 $P \leftarrow$ processor whose context is on top of memory
$s \leftarrow$ superstep number to be simulated next for $P$
$C \leftarrow i_s$-cluster containing $P$
2 Simulate superstep $s$ for $C$
3 **if** $P$ has finished its program **then exit**
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 $C = \hat{C}_j$ for some index $j$
4.1 **if** $j > 0$ **then** swap the contexts of $C$ with those of $\hat{C}_0$
4.2 **if** $j < 2^{i_s - i_{s+1}} - 1$ **then**
swap the contexts of $\hat{C}_0$ with those of $\hat{C}_{j+1}$

**Figure 1. Structure of the simulation scheme.**

The correctness of the resulting strategy was proved in [13]. Note that the simulation brings new clusters to the top of memory only when $i_{s+1} < i_s$ since, in order to simulate superstep $s+1$ for an $i_{s+1}$-cluster $\hat{C}$, superstep $s$ must have been executed for all of the $i_s$-clusters contained in $\hat{C}$. Instead, if $i_{s+1} \geq i_s$, no cluster swaps are performed, and the next round will simulate superstep $s+1$ for the topmost $i_{s+1}$-cluster that is contained within the $i_s$-cluster $C$ currently residing on top of memory. Advancing the simulation unevenly for the different D-BSP submachines proved crucial in [13] to transform the submachine locality exhibited by $\mathcal{P}$ into temporal locality. However, in order to exploit also spatial locality effectively, as encouraged by the BT model, the actual implementation of the above scheme requires major modifications over the one in [13]. Subsection 3.1 describes a suitable memory layout needed by such an implementation; a detailed implementation of step 2 will then be presented in subsection 3.2; finally, the running time of the simulation will be determined in subsection 3.3.

### 3.1. Memory Layout

Although the simulation algorithm of [13] yields a valid BT program, it is not designed to exploit block transfer. For example, during the simulation of local computation, the scheme brings one context at a time to the top of memory, which is highly inefficient in the BT framework. As suggested in [2] a good BT algorithm must be recursive, and block transfer must be used at every level of recursion. Since the BT model supports block copy operations only for nonoverlapping 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 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**
Shift blocks $v/2^{i+1}, \ldots, v/2^i - 1$
    to blocks $v/2^i, \ldots, 3v/2^{i+1} - 1$
UNPACK($i + 1$)

Note that the shift operation executed by UNPACK can be performed as 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 followed $C$ among the contexts of $C$ itself. figure 2 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 polynomially bounded, 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 topmost $i$-cluster. Clearly, the same complexity applies to PACK($i$). Thus, the running times of the two subroutines are

$$T_{\mathrm{UNPACK}}(i), T_{\mathrm{PACK}}(i) = O\left(v/2^i\right). \qquad (1)$$

In order to create the required buffer space for swaps during the simulation, we augment the code presented in the previous section as shown in figure 3. Observe that the augmented 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).

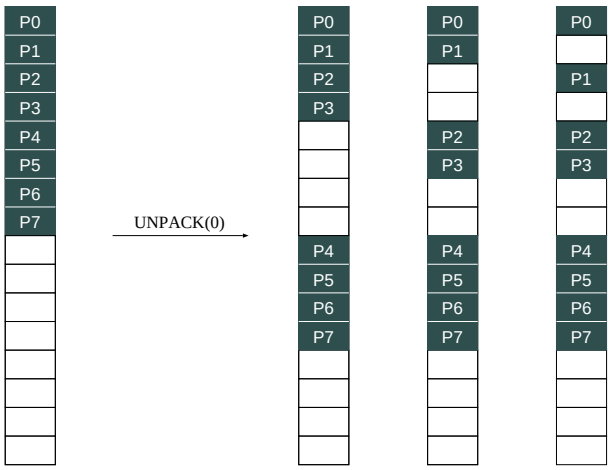


**Figure 2. Snapshots of the BT memory layout during an UNPACK(0) operation. The host D-BSP machine has 8 virtual processors. Solid boxes indicate processor contexts, and white boxes indicate empty buffers.**

0 UNPACK(0)
  **while** true **do**
1  $P \leftarrow$ processor whose context is on top of memory
  $s \leftarrow$ superstep number to be simulated next for $P$
  $C \leftarrow i_s$-cluster containing $P$
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**
    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 $j$
4.1  **if** $j > 0$ **then** swap the contexts of $C$ with those of $\hat{C}_0$
4.2  **if** $j < 2^{i_s - i_{s+1}} - 1$ **then**
      swap the contexts of $\hat{C}_0$ with those of $\hat{C}_{j+1}$
5  UNPACK($i_s$)

**Figure 3. The revised simulation scheme.**

### 3.2. Simulation of a superstep

The simulation of superstep $s$ for an $i_s$-cluster $C$ (step 2 of the algorithm presented before) is performed 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.** Thanks to the invariant mentioned before and by step 1.a of the algorithm in Figure 3, 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, processors' 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. To specify the chunk size we need the following definition.

**Definition 1** *Let $n$ be a power of 2. Define $c(n)$ as the greatest power of 2 such that $c(n) \leq \min\{f(\mu n)/\mu, n/2\}$.*

Clearly, we have that $c(n) > (1/2)\min\{f(\mu n)/\mu, n/2\}$. The local computation of superstep $s$ for $C$ is simulated by invoking the recursive function COMPUTE($n$), whose code is given in figure 4, with $n = v/2^{i_s}$.

```
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 to blocks c, ..., n − 1
```

**Figure 4. The COMPUTE subroutine.**

The correctness of COMPUTE($n$) can be proved by induction, based on 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 block of memory cells, 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 memory swaps. 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

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

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

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

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

$$T_{\mathrm{COMP}}(C, s) = \frac{v}{2^{i_s}}\tau + O\left(\frac{v}{2^{i_s}} \log\log \frac{v}{2^{i_s}}\right), \quad (2)$$

where $\tau$ denotes 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, suppose that a processor $P_j$ exchanges messages through a unified input/output queue which resides at the end of $P_j$'s context. To deliver all messages to their destinations, we make use of sorting. Specifically, the contexts of $C$ are divided into $\Theta(|C|)$ 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 in the queue at the end of $P_j$'s context. This is easily achieved by sorting elements according to suitably chosen tags attached to the elements[1].

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 $\Theta\left(v/2^{i_s}\right)$ so the required memory space is

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

[1]The required tagging can be produced during the simulation of local computation without asymptotically increasing the running time. Full details will be provided in the final version of the paper.

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 $0 \leq i \leq i_s$, 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 $\mu v/2^{i_k} \geq L(i_s)$, or 0 if $\mu v < L(i_s)$. Then, we can free a sufficient amount of space for sorting through the following steps.

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 $O(L(i_s))$ 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 input queue after sorting is proportional to the amount of data received by the processor, and each processor may receive a different amount of data. 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
  Shift 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($n/2$)
  Move blocks $0, \ldots, n/2 - 1$ to blocks $n/2, \ldots, n - 1$
  Move blocks $n, \ldots, 3n/2 - 1$ to 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 $O(n \log n)$, 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

$$T_{\text{COMM}}(C, s) = O\left(\frac{v}{2^{i_s}} \log \frac{v}{2^{i_s}}\right). \tag{3}$$

### 3.3. Analysis of the simulation time

By adding up the contributions of the local computation and communication phases (equations 2 and 3, respectively) we conclude that the simulation of superstep $s$ for an $i_s$-cluster $C$ requires time

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

It is important to observe that this bound *does not* depend on $g(x)$ or $f(x)$, as long as $f(x) = O(x^\alpha)$, for some constant $0 < \alpha < 1$. Moreover, we remark that, besides the unavoidable term $(v/2^{i_s})\tau$, the sorting time is the dominant factor.

If $i_{s+1} < i_s$, the simulation algorithm incurs an additional cost due to the need of moving a new $i_s$-cluster to the top of memory, as prescribed by steps 4.1 and 4.2 of the pseudo-code presented in figure 3. 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 two block transfers taking time $O(f(j|C|) + |C|)$. 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

$$O\left(2^{i_s - i_{s+1}} f\left(\frac{v}{2^{i_{s+1}}}\right) + \frac{v}{2^{i_{s+1}}}\right). \tag{4}$$

Assume now that for any two consecutive supersteps $s$ and $s+1$ in $\mathcal{P}$, with $i_{s+1} < i_s$, the hypothesis $f(v/2^{i_{s+1}}) \leq v/2^{i_s}$ is satisfied. If this is the case, we can immediately conclude that the overhead of cluster swaps is $O\left(v/2^{i_{s+1}}\right)$, hence it is amortized by the cost of the future simulation of the $i_{s+1}$-cluster. As a consequence, the cost of step 4 of the simulation 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 $\mathcal{P}$. As shown below, the hypothesis on consecutive superstep indexes can be removed without asymptotically affecting the running time of the simulation. Thus, we obtain the following general result.

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

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

*Proof.* Before the simulation begins, $\mathcal{P}$ is augmented by inserting the minimum number of *dummy supersteps* to enforce the property that any two consecutive supersteps $j$ and $j+1$, with $i_{j+1} < i_j$ in $\mathcal{P}$ satisfy the inequality

$$f(v/2^{i_{j+1}}) \leq v/2^{i_j}, \tag{5}$$

so that the complexity analysis performed at the beginning of this subsection holds. Consider now two "original" supersteps $s$ and $s+1$ of $\mathcal{P}$ with $i_{s+1} < i_s$ which do not satisfy inequality 5. The number $k$ of dummy supersteps to be inserted is such that $f^{(k)}(v/2^{i_{s+1}}) \leq v/2^{i_s}$, whence $k \leq f^*(v/2^{i_{s+1}}) = O\left(\log\log(v/2^{i_{s+1}})\right)$.

With regard to the simulation, the scheme we described in subsection 3.2 is slightly modified to treat dummy and nondummy supersteps differently: to be precise, the simulation algorithm skips step 2 for dummy supersteps. 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 4). This cost is given by equation (4) and is

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

where $i_{j+1} < i_j$. Since $i_j$ and $i_{j+1}$ now satisfy inequality (5), the cost for the simulation of the dummy $i_j$-superstep for all $i_j$-clusters is $O\left(v/2^{i_{j+1}}\right)$. Based on this result, we can say that for every $i_{s+1}$-cluster, the extra cost for dealing with the $O\left(\log\log(v/2^{i_{s+1}})\right)$ dummy supersteps which are placed between nondummy superstep indexes $i_s$ 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 cost of the simulation of the $i_{s+1}$-superstep. □

## 4. Discussion and Applications

Theorem 1 shows that the simulation algorithm is quite efficient. Indeed, Proposition 1 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. For concreteness, consider the case of *matrix multiplication*. On D-BSP $(v, O(1), g(x))$, the algorithm of [9, 13] multiplies two $\sqrt{v} \times \sqrt{v}$ matrices by using $2^i$ $2i$-supersteps, for every $0 \leq i < \log(v)/2$, performing constant local computation per superstep. It is easy to see that our simulation of this algorithm yields an optimal $O\left(v^{3/2}\right)$ algorithm for $f(x)$-BT, while a trivial step-by-step D-BSP simulation would have required at least time $\Omega\left(v^{3/2}\log^* v\right)$ for $f(x) = \log x$, and time $\Omega\left(v^{3/2}\log\log v\right)$ for $f(x) = x^\alpha$.

Although the exploitation of submachine locality was already featured by the simulation algorithm in [13], the efficiency of that algorithm crucially relied on the fact that the cost function $g(x)$ of the guest D-BSP machine was *the same* as the access function $f(x)$ of the host HMM machine. In contrast, for the simulation presented in this paper, the running time is independent of the actual values of these two functions. 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 $\log x$-BT and $x^\alpha$-BT.

A natural deployment of our result is obtaining sequential algorithms that exploit both temporal and spatial locality by simulating fine-grained D-BSP algorithms. However, different D-BSP cost functions may require different algorithmic strategies, and without a strict correlation between the D-BSP cost function $g(x)$ and the BT access function $f(x)$ in the simulation, the question arises of which function $g(x)$ suggests the best "coding practices" for $f(x)$-BT.

Unlike the HMM scenario [13], the choice $g(x) = f(x)$ is not always the best. Consider, for instance, the problem of computing the *Discrete Fourier Transform* of $v$ points ($v$-DFT). Two D-BSP algorithms for this problem are applicable: the first algorithm is a standard execution of the $v$-input FFT dag, while the second is based on a recursive decomposition of the same dag into two layers of $\sqrt{v}$ independent $\sqrt{v}$-input subdags, which are assigned to distinct clusters of the parallel machine, and are separated by a transpose permutation requiring a 0-superstep. On D-BSP $(v, O(1), x^\alpha)$, both algorithms yield a running time of $O\left(v^\alpha\right)$ [13], which is optimal. However, it is easy to see that the simulation times of these two algorithms on the $x^\alpha$-BT are $O\left(v\log^2 v\right)$ and $O(v\log v\log\log v)$, respectively. This implies that the choice $g(x) = f(x)$ is not *effective* [7], in the sense that D-BSP $(v, O(1), x^\alpha)$ does not reward the use of the second algorithm over the first. On the other hand, it can be shown that D-BSP $(v, O(1), \log x)$ correctly distinguishes among the two algorithms, since their respective parallel running times are $O\left(\log^2 v\right)$ and $O(\log v\log\log v)$.

The above example is a special case of the following more general consideration. Observe that Theorem 1 yields a linear-slowdown simulation 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 $(v, O(1), \log x)$.

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 permu-

tations on $f(x)$-BT can be employed to design a D-BSP program 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 $O(\log v \log\log v)$-time algorithm for the $v$-DFT designed for D-BSP $(v, O(1), \log x)$. 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 $O(v \log v)$, 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 nonoptimality 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).

## References


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

[2] A. Aggarwal, A. K. Chandra, and M. 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.

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

[4] A. Bäumker, W. Dittrich, and F. 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.

[5] G. Bilardi, K. Ekanadham, and P. 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.

[6] G. Bilardi, C. Fantozzi, A. Pietracaprina, and G. 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.

[7] G. Bilardi, A. Pietracaprina, and G. 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.

[8] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. 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.

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

[10] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 106–115, 1997.

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

[12] F. Dehne, D. Hutchinson, A. Maheshwari, and W. Dittrich. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proceedings of the 13th International Parallel Processing Symposium*, pages 14–20, Puerto Rico, 1999.

[13] C. Fantozzi, A. Pietracaprina, and G. 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.

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

[15] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 285–298, October 1999.

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

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

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

[19] U. Vishkin. Can parallel algorithms enhance serial implementation? In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 376–385, Cancún, Mexico, April 1994. IEEE Computer Society.

[20] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[21] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2-3):148–169, 1994.