Optimizing Parallel Bitonic Sort

by

MIHAI FLORIN IONESCU

A thesis submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at SANTA BARBARA

Committee in charge:

Professor Klaus E. Schauser, Chair
Professor Martin C. Rinard
Professor Ambuj K. Singh

August 1996

The thesis of MIHAI FLORIN IONESCU is approved:

Chair Date

Date

Date

University of California at Santa Barbara

August 1996

Pentru părinţii mei, Florin şi Marioara.

MIHAI FLORIN IONESCU

Department of Computer Science
University of California, Santa Barbara

Optimizing Parallel Bitonic Sort

Abstract

Sorting is an important component of many applications and parallel sorting algorithms have been studied extensively in the last three decades. One of the earliest parallel sorting algorithms is Bitonic Sort which is represented by a sorting network consisting of multiple butterfly stages. Not surprisingly, the algorithm has been studied on different parallel network topologies which provide an easy embedding of butterflies, such as the hypercube or the shuffle-exchange.

This thesis studies bitonic sort on modern parallel machines. These machines are relatively coarse grained and consist of only a modest number of nodes. Thus many data elements have to be mapped onto each processor. Under such a setting optimizing the bitonic sort algorithm becomes a question of mapping the data elements to processing nodes (data layout) to minimize communication and optimizing the local computation on each node. We developed a bitonic sort algorithm which minimizes the number of communication steps and optimizes the local computation. We have implemented the new algorithm in Split-C, analyzed and evaluated its performance under the LogP and LogGP models of parallel computation. The resulting algorithm is faster than the previous implementations, as experimental results collected on a 64 node Meiko CS-2 show.

Contents

List of Figures

List of Tables

Chapter 1

The Parallel Sorting Challenge

1.1 Overview

Sorting is a popular Computer Science topic which received more attention than most any other problem. [Knu73], for instance, found that computers devote approximately a quarter of their time to sorting. In the last decades parallel processing has added a new dimension to the sorting problem and stimulated the research on sorting algorithms. Parallel sorting is present in a wide variety of practical applications and many parallel sorting algorithms have been studied extensively for a variety of parallel models of computation. The problem is especially interesting because it fundamentally requires communication as well as computation, being an excellent area to investigate the effect of communication and computation on algorithm performance [ABK95]. One of the earliest parallel sorting algorithms is Bitonic Sort [Bat68], which is represented by a sorting network connected by multiple butterfly stages of increasing size. Since then, a wide range of sorting networks have been proposed, and their complexity and performance have been studied in great detail. The bitonic sort was the first network capable of sorting n elements in $O(\lg^2 n)$ time. Not surprisingly, bitonic sort has been studied extensively on parallel network topologies such as the hypercube and shuffle-exchange which provide an easy embedding of butterflies [Sto71]. Various properties of bitonic networks have been investigated, e.g.

[Knu73, HS82, BN86]. Recent implementations and evaluations of parallel algorithms show that although bitonic sort is slow for large data sets (compared for example with radix sort or sample sort) it is more space-efficient and represents one of the fastest alternatives for small data sets (see [CDMS94, BLM+91]).

Earlier parallel sorting algorithms have being studied in the context of PRAM or network-based models. In both approaches, algorithms were developed under the assumption that the number of processors (P) is comparable to the number of data elements (N). In real applications, however, the number of data elements is much greater than the number of processors ($N \gg P$) and therefore most of the algorithms proposed for bitonic sort performed poorly in practice because the parallel models under which they were designed are not realistic enough for today's parallel machines.

Parallel sorting is only one example of a parallel application for which the transition from a theoretical model to an efficient implementation is not straightforward. Most of the research on parallel algorithm design in the '70s and '80s has focused on fine-grain models of parallel computation (see [BDHM84, Já92, KR90, Lei92, Rei93, Qui94, KGGK94]) where the ratio of memory to processors is relatively small. Later research has shown, however, that processor-to-processor communication is the most important bottleneck in parallel computing (see [ACS90, CKP+93, KRS90, PY88, Val90a, Val90b]). Therefore efficient parallel algorithms are more likely to be achieved on coarse-grain parallel systems and in most situations the original algorithm is substantially redesigned (see [AISS95]).

As a parallel application, sorting is especially challenging because of the amount of communication it requires. Essentially, almost all data items may move from the processor they reside originally on to some other processor. By minimizing the number of data communication steps, the total volume of data transfered and the number of messages sent across the network, we are likely to obtain algorithms that perform efficiently in practice.

1.2 Bitonic Sort on Parallel Computers

In the case of parallel bitonic sort, in order to achieve the $O(\lg^2 n)$ time bound, the algorithm assumes that each node of the bitonic sorting network is mapped onto a separate processor and that connected processors can communicate in unit time. The network size will grow proportionally to the input size. As noted above modern parallel machines, however, have generally a high communication overhead and are much coarser grained, consisting of only a relatively small number of nodes. Thus many data elements have to be mapped onto each processor. One of the interesting problems here is how we can exploit the grouping of data among processors. Generally we can distinguish three different phases for a parallel algorithm and in particular a sorting algorithm:

- a purely local computational phase,
- an optional intermediate phase which calculates the destination of elements for the next phase,
- a communication phase which moves keys across processors.

Under such a setting optimizing a parallel algorithm becomes a question of optimizing communication as well as computation. This involves a strategy of mapping the data elements to processing nodes in such a way that communication requirements are minimized and the local computation on each node is optimized.

This thesis applies these techniques to parallel bitonic sort and optimizes both communication and computation. We derive a new data layout which allows us to perform the smallest possible number of data remaps. Compared with previous approaches our algorithm executes less communication steps and transfers less data. Furthermore, by taking advantage of the special format of the data input, we show how to optimize the local computation on each node.

We develop an efficient implementation of our algorithm in Split-C [CDG+93], a simple parallel extension of the language C. Split-C provides a machine independent

SPMD programming model and is supported on a large variety of parallel architectures. Experimental results, collected on a 64 node Meiko CS-2, show that optimizing the communication obtains the most dramatic improvement. For the Meiko CS-2, which has special support for long messages [SS95], we analyze the impact of long messages on the execution time and on the algorithm design and implementation.

We also investigate the factors that influence communication in a remap-based parallel bitonic sort algorithm by analyzing the algorithm under the framework of realistic models for parallel computation such as LogP [CKP+93] or LogGP [AISS95].

Finally we compare our implementation of bitonic sort against other parallel sorts.

1.3 Contributions

This thesis makes the following contributions:

- It develops a parallel bitonic sort algorithm with the smallest number of data remaps.

- It develops an $O(\log n)$ time algorithm for finding the minimum of a bitonic sequence with no duplicate elements.

- It shows how to optimize the local computation part assigned to each processor, by replacing the simulation of the compare-and-swap nodes of the bitonic network with much faster sorting and bitonic merge routines.

- It presents experimental results, collected on a Meiko CS-2, which show that both optimizing the communication and computation results in a substantial performance improvement over previous approaches.

- It analyzes communication in remap-based parallel algorithms under various circumstances and with respect to different communication efficiency metrics.

- It compares the algorithm against other parallel sorting algorithms, and presents experimental results which show that under certain circumstances, e.g. on a small number of processors, our parallel bitonic sort actually performs better than other parallel sorts.

The remainder of the thesis is structured as follows. Chapter 2 describes the parallel bitonic sort algorithm and the bitonic sorting network and discusses basic data layouts used in naive parallel implementations of bitonic sort. Chapter 3 derives an optimal data layout which minimizes the number of data remap steps. Chapter 4 discusses how to optimize the local computation on each processor. Chapter 5 presents experimental results collected on a 64 processor Meiko CS-2. Chapter 6 discusses related work, and Chapter 7 concludes and presents future work.

Chapter 2

Bitonic Sort

In this section we describe the bitonic sort algorithm originally developed by Batcher three decades ago [Bat68]. After introducing the basic definitions and discussing the algorithm, we present a naive implementation of the parallel bitonic sort and discuss possible data layouts for speeding up the sorting process.

2.1 Basic Algorithm and Definitions

Bitonic sort is based on repeatedly merging two *bitonic sequences* to form a larger bitonic sequence. The following basic definitions were adapted from [KGGK94].

2.1.1 Bitonic Sequence

Definition 1 (Bitonic Sequence) *A* bitonic sequence *is a sequence of values* $a_0, \ldots, a_{n-1}$*, with the property that (1) there exists an index* i*, where* $0 \leq i \leq n-1$*, such that* a_0 *through* a_i *is monotonically increasing and* a_i *through* a_{n-1} *is monotonically decreasing, or (2) there exists a cyclic shift of indices so that the first condition is satisfied.*

For example, $\langle 2,3,4,5,6,7,8,8,7,5,3,2,1 \rangle$ is a bitonic sequence because it first increases and then decreases. Similarly, $\langle 6,7,8,8,7,5,3,2,1,2,3,4,5 \rangle$ is also a bitonic sequence, because it is a cyclic shift of the first bitonic sequence as shown in Figure 2.1.

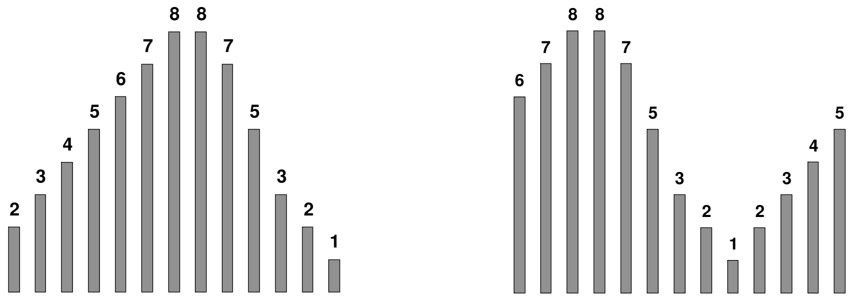


Figure 2.1: *Bitonic sequences can have different shapes. The sequence on the right is a cyclic shift of the sequence on the left.*

2.1.2 Bitonic Split and Bitonic Merge

Definition 2 (Bitonic Split) *Consider a bitonic sequence $s = a_0, a_1, \ldots, a_{n-1}$, where n is a power of 2 and the following subsequences of s:*

$$\begin{aligned} s_1 &= \langle \min(a_0, a_{n/2}), \min(a_1, a_{n/2+1}), \ldots, \min(a_{n/2-1}, a_{n-1}) \rangle \\ s_2 &= \langle \max(a_0, a_{n/2}), \max(a_1, a_{n/2+1}), \ldots, \max(a_{n/2-1}, a_{n-1}) \rangle \end{aligned}$$

The above operation is known as a bitonic split.

A bitonic split operation has two fundamental properties (for details see [KGGK94]):

1. s_1 and s_2 are bitonic sequences.

2. the elements of s_1 are all smaller than the elements of s_2.

Thus, given a bitonic sequence we can recursively obtain shorter bitonic sequences using the bitonic split transformation described above, until we obtain sequences of size one. At this point the input sequence is sorted. Therefore $\lg n$ splits are required to sort a bitonic sequence of size n. This procedure of sorting a bitonic sequence using bitonic splits is called *bitonic merge* and it is easy to implement on a network of comparators. This network of comparators, known as a *bitonic merging network* is illustrated in Figure 2.2 for an input of size $N = 8$.

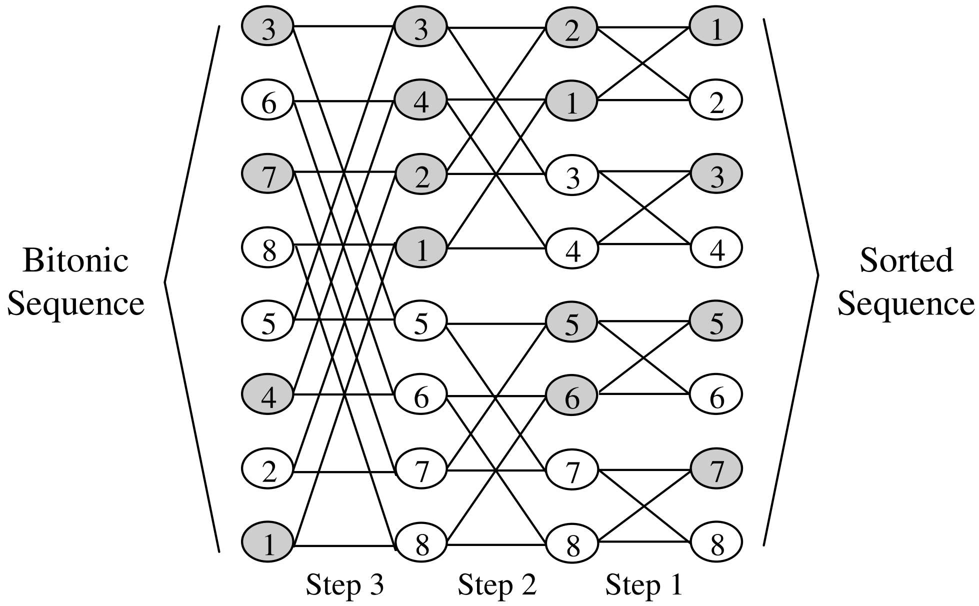


Figure 2.2: *Butterfly structure of an increasing bitonic merge of size $N = 8$ ($BM_8^{\oplus}$). A shaded node designates an address where the minimum of the two keys is placed, the unshaded node designates an address where the maximum of the two keys is placed.*

2.1.3 Bitonic Sorting Network

We now introduce the bitonic sorting network which is used to implement the bitonic sort algorithm:

Definition 3 (Bitonic Sorting Network) *The* bitonic sorting network *for sorting N numbers consists of* $\lg N$ bitonic sorting stages*, where the i-th stage is composed of $N/2^i$ alternating increasing and decreasing* bitonic merges *of size 2^i (see Figure 2.3).*

The communication structure of a bitonic merge of size 2^i is represented by a butterfly with 2^i rows and $i+1$ columns (Figure 2.2), where each butterfly node selects the minimum or the maximum of the two inputs.

Each node of the bitonic sorting network is identified by a 3-tuple (s, c, r), where the three elements are the stage*, the* column *inside the stage and the* row *of the node, respectively. Stage s contains $s+1$ columns numbered $s, \ldots, 0$. Column 0 of stage s is called the output of stage s and corresponds also to column $s+1$ of stage $s+1$ which is called the input of stage $s+1$. The transition from column i to column $i-1$ is called* step i *(see Figure 2.4).*

The connectivity of the network is described by the following relation: the node (s, c, r), where $1 \leq s \leq \lg N$, $0 \leq c \leq s-1$ and $0 \leq r < N$, receives inputs from nodes $(s, c+1, r)$ and $(s, c+1, \overline{r_c})$, where $\overline{r_c} = r \oplus 2^c$ (i.e. r and $\overline{r_c}$ differ only in bit c). The network has two types of nodes, MIN and MAX. The node (s, c, r), where $0 \leq c \leq s-1$, selects the minimum of the two inputs if $(r\ div\ 2^c) \bmod 2 = (r\ div\ 2^s) \bmod 2$, otherwise it selects the maximum.

What is not obvious from the *Bitonic Sorting Network* definition is that this network actually sorts the input. This results, however, from the duality of the *network view* and the *algorithmic view* of bitonic sort.

The *network view* was introduced by the previous definition. It presents the overall structure of the bitonic sorting network and describes the function performed by each of its components. The dual view is the *algorithmic view*. Under this abstraction we view each column of the network as an array containing all of the data elements.

The primitive operation is a *compare-exchange* which compares two numbers whose addresses differ in only one bit and exchanges them, if necessary, so that they are in the proper order. Using this primitive we can implement a bitonic split and a bitonic merge, and show that the algorithm actually sorts the input (for details we refer the interested reader to [KGGK94]).

Since for the rest of the thesis we make extensive use of the *network view* of bitonic sort, we focus especially on it and describe its components in more detail.

A *bitonic merge* of size k ($BM_k^{\oplus}$ or $BM_k^{\ominus}$) takes as *input* a bitonic sequence of length 2^k and generates as *output* a sorted sequence in increasing or decreasing order, depending on its type ($\oplus$ or $\ominus$, see Figure 2.2 and Figure 2.3). During the i-th bitonic sorting stage $N/2^i$ monotonic ordered sequences of length 2^i are formed from bitonic sequences of length 2^{i-1}. The monotonic sequences are ordered such that two neighboring sequences (one monotonically increasing and one monotonically decreasing) can be combined to form a new bitonic sequence for the next bitonic merge (see Figure 2.3).

The j-th step of each stage (notice that steps are counted from right to left) takes as input $N/2^j$ bitonic sequences of length 2^j and, using compare-exchange operations between pairs of nodes whose addresses differ only in the j-th bit, halves each bitonic sequence into two bitonic sequences such that every element of the first sequence is smaller than every element of the second sequence, operation described previously and known as bitonic split.

The communication structure of the bitonic sorting network can be visualized as the concatenation of increasingly larger butterflies. Figure 2.4 presents a bitonic sorting network of size $N = 8$. There are $\lg N = 3$ merge stages and the i-th merge stage consists of i *merge steps*. Figure 2.3 shows the block structure of the bitonic sorting network. With $BM_k^{\oplus}$ we have denoted the increasing bitonic merging networks of size k, and with $BM_k^{\ominus}$ the decreasing merging networks.

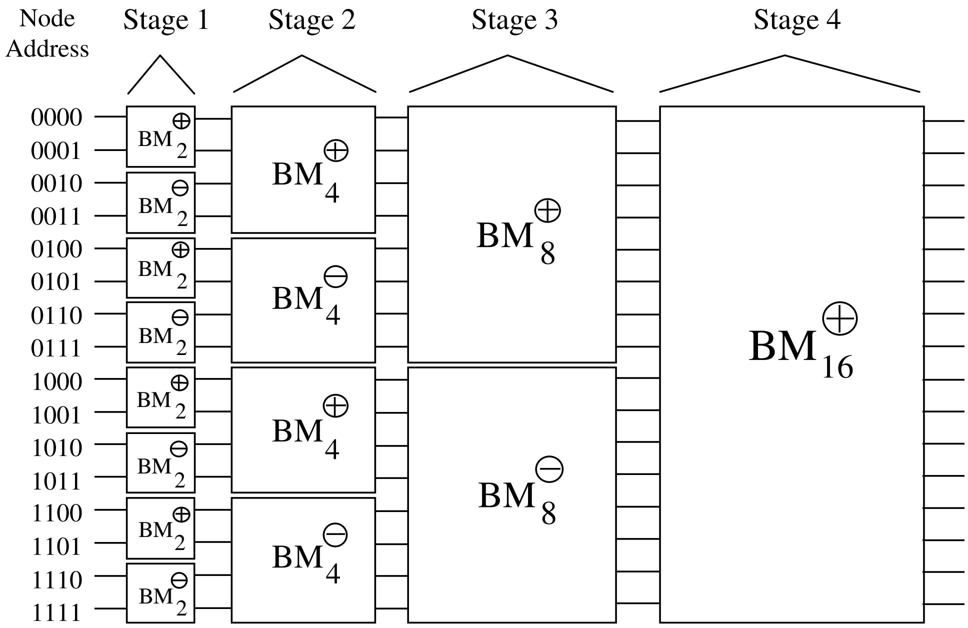


Figure 2.3: *Block structure of a bitonic sorting network of size* $N = 16$. *With* $BM_k^{\oplus}$ *and* $BM_k^{\ominus}$ *we denote increasing, respectively decreasing, bitonic merging networks of size* k *(From [KGGK94]).*

For the rest of the thesis we consider N to be the data size, P the number of processors and $n = N/P$ the number of elements which go on one processor. Because the number of nodes in the bitonic sorting network is a power of two, we also assume N and P to be powers of two.

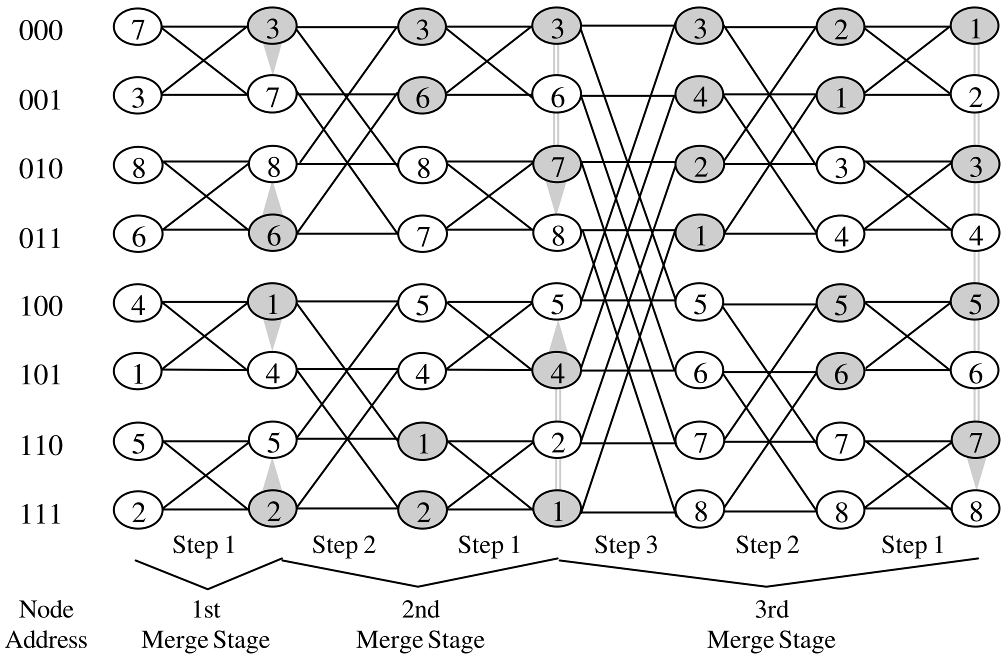


Figure 2.4: *A bitonic sorting network of size $N = 8$. A row of nodes represents an address containing one of the keys. Each node compares two keys, as indicated by the edges and selects either the maximum or the minimum. A shaded node designates an address where the minimum of the two keys is placed, the unshaded node designates an address where the maximum of the two keys is placed. The arrows indicate the monotonic ordered sequences, with the arrowhead pointing towards the largest key (From [CDMS94]).*

2.2 Naive Parallel Implementation

As presented above the bitonic sorting network consists of a series of butterfly networks of increasing size, where each butterfly node represents a comparison operation. For maximal parallelism, it would be desirable to have a separate processor unit available for each row of nodes and an interconnection network whose connectivity reflects the bitonic sorting network. In reality the size of the input is substantially larger than the number of processors and thus many data items have to be mapped to each processor.

There are two optimization steps which need to be done under such a case. First, a data layout has to be found which minimizes *communication*. Second, the *computational* phase mapped to each processor must be rearranged such that it is optimized. The *data layout* determines when and where communication occurs and also which processor is responsible for which computation.

The naive approach represents one straightforward parallel implementation: simply simulate the compare-exchange steps in the butterfly network using a *blocked data layout.*

Definition 4 (Blocked Layout) *A* blocked layout *for mapping N keys on P processors assigns the i-th key to the $\lfloor i/n \rfloor$-th processor, where $n = N/P$.*

Therefore, under a blocked layout, the first N/P keys are assigned to the first processor, the second N/P keys to the second processor and so on. N represents the data size, P represents the number of processors; on each processor we will have $n = N/P$ elements. Figure 2.5 presents a blocked data layout for $N = 16$ elements and $P = 4$ processors and highlights the communication which occurs under such a layout. Communication occurs when two nodes connected by an arc are located on different processors. The black arcs highlight the remote accesses, while the grey arcs indicate local accesses. Under a blocked layout, the first $\lg n$ stages execute completely local. For a subsequent stage $\lg n + k$, where $1 \leq k \leq \lg P$, the first k steps compare nodes that are located on different processors (therefore requiring communication) while the last $\lg n$ steps are completely local. Another possible data layout is the *cyclic layout*:

Definition 5 (Cyclic Layout) *A* cyclic layout *for mapping N keys on P processors assigns the i-th key to the $(i \bmod n)$-th processor, where $n = N/P$.*

Under a cyclic layout, the first key (i.e., the first row of nodes) is assigned to the first processor, the second key (and second row of nodes) to the second processor and so on. Figure 2.6 shows the communication which occurs under a cyclic layout (black arcs highlight the remote accesses and grey arcs indicate local accesses). Compared to the blocked layout just the reverse happens: the first $\lg n$ stages require remote accesses. For subsequent stages $\lg n + k$, where $1 \leq k \leq \lg P$, the first k steps are completely local, while the last $\lg n$ steps require remote communication. Overall a cyclic layout has a higher communication complexity than a blocked layout. As we can see communication is strongly affected by the data layout.

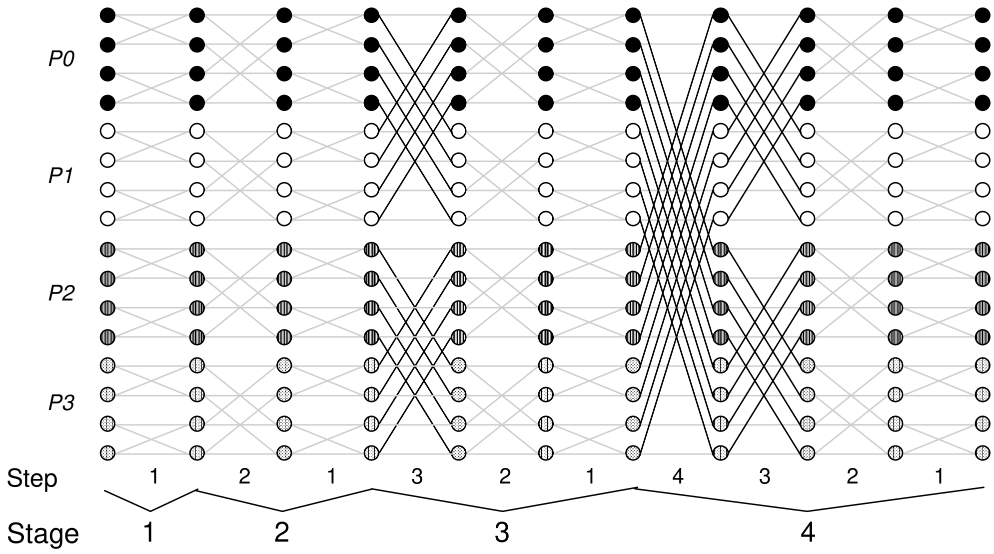


Figure 2.5: *A blocked data layout for a bitonic sorting network for 16 elements on 4 processors. The shading of the nodes reflects the allocation onto the processors. Black arcs indicate remote communication, while grey arcs indicate local accesses.*

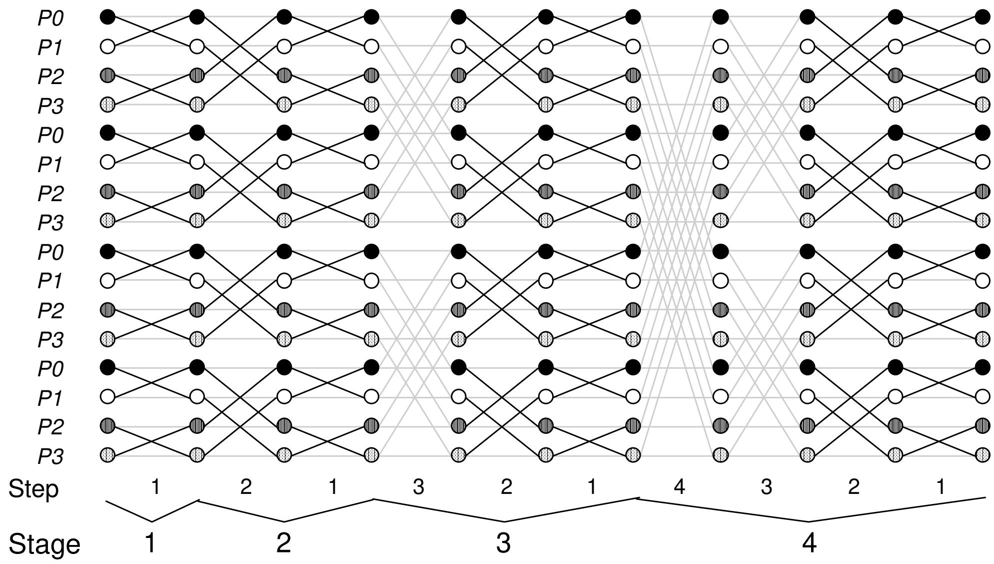


Figure 2.6: *A cyclic data layout for a bitonic sorting network for 16 elements on 4 processors. The shading of the nodes reflects the allocation onto the processors. Black arcs indicate remote communication, while grey arcs indicate local accesses.*

2.3 Improving the Data Layout

One efficient data placement which minimizes the communication requirements is to use different data layouts so that all compare-exchange operations execute locally.

Since under a cyclic layout the first k steps of the stage $\lg n + k$ are completely local, we can reduce the communication requirements by periodically remapping the data from a blocked layout to a cyclic layout and vice versa. This approach was first suggested in [CKP+93, CDMS94] and used for efficient implementations of parallel algorithms based on the butterfly network such as FFT [CKP+93] or bitonic sort [CDMS94].

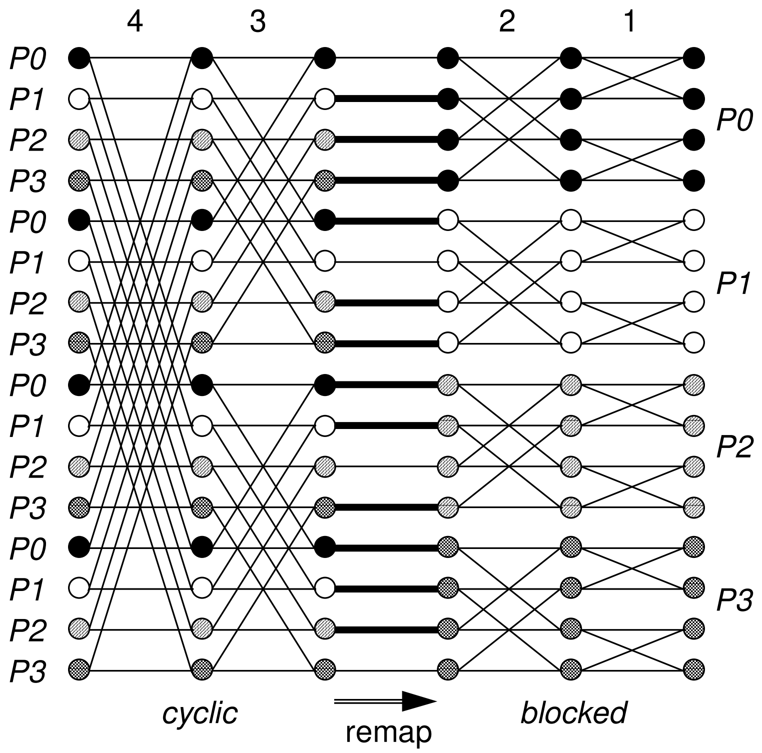


Figure 2.7: *Remap from a cyclic to a blocked layout ($N = 16$ nodes, $P = 4$ processors). Thick arcs highlight where communication occurs, normal arcs indicate local accesses.*

Under this remapping strategy the algorithm starts with a blocked layout. Therefore, the first $\lg n$ stages are entirely local. For each subsequent stage $\lg n + k$, where $1 \leq k \leq \lg P$, we remap from a blocked to a cyclic layout, compute the first k steps locally, remap back into a blocked layout, and perform the last $\lg n$ steps locally. Thus we have reduced the communication requirements to only two remap operations per stage for the last $\lg P$ stages. Figure 2.7 shows a cyclic to blocked remap for the last stage of a $N = 16$ node and $P = 4$ processors bitonic sorting network. Each remap operation involves an all-to-all communication of n words per processor, as opposed to a fixed data layout where each remote step involves exchanging n words between pairs of processors. The former can be implemented under the LogP [CKP+93] or LogGP [AISS95] models nearly as efficiently as one-to-one communication.

Periodic cyclic-blocked remapping has some limitations, however: it is dependent on the data size. In order to do a remap from cyclic to blocked across a stage and execute compare-exchange operations locally we need to have $N \geq P^2$, i.e., at least P elements on each processor. A brief justification is that both the blocked and the cyclic layout can each cover at most $\lg \frac{N}{P}$ steps, thus for the final stage we must have $\lg N \leq 2 \lg \frac{N}{P}$ which implies $N \geq P^2$. We will show in more detail why this is the case in the next chapter.

The remapping strategy presented above was to our knowledge the most efficient technique used so far for the parallel bitonic sort (for implementation details and experimental results see [CDMS94]). The following chapters will show how an improved communication-efficient data layout and remapping strategy can be derived and how local computation can be optimized.

Chapter 3

Optimizing Communication

In this chapter we discuss how to minimize the communication requirements. The next chapter addresses how to optimize the local computation on each processor.

As we saw from the cyclic-blocked implementation a good data distribution can dramatically reduce the communication requirements. Since today's parallel machines have a high communication overhead, it is very important to keep communication as low as possible. The straightforward observation is that, in order to keep the communication cost low, merging steps need to be local, i.e., elements which are compared and eventually swapped should be on the same processor. Therefore the algorithm should consist of a series of local computation phases and data remaps.

The communication complexity of a remap based algorithm can be analyzed with respect to different metrics such as *total volume of data transfered*, *the number of communication steps* (i.e. remaps) or *total number of messages sent*. A specific algorithm can be optimal with respect to some of these metrics, however the total communication time depends on all of them. In one of the following sections we will analyze the communication requirements of different versions of bitonic sort under the LogP and LogGP models which allow us to compute the communication complexity of the algorithm with respect to all of the above metrics. The main result presented in this chapter is that we derive an algorithm which executes the smallest possible number of data remaps and therefore we will minimize the number of communication steps.

3.1 Absolute and Relative Addresses

In this section we introduce some new notation:

Definition 6 (Absolute and Relative Address of a Node) *The* absolute address *of a node ($\lg N$ bits long), in the bitonic sorting network, is represented by the number of the row where the node was initially mapped.*

After a remap operation a node will have a relative address. *This consists of two parts (see Figure 3.1):*

- *The processor number where the node is remapped ($\lg P$ bits long).*
- *The local address on the processor ($\lg n$ bits long).*

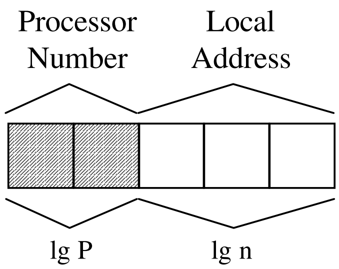


Figure 3.1: *Bit representation of a node's relative address for $N = 32$ nodes and $P = 4$ processors. The shaded area represents the* processor part *of the address.*

The way the remap operation is actually implemented is by computing the relative address of each node for the new data layout. After we perform the remap (i.e. we change the data layout) each node will be uniquely identified by its absolute and relative address.

It is natural to start the algorithm with a blocked data layout because we can perform the first $\lg n$ stages of the bitonic sorting network entirely local. Then we can reduce the communication requirements by remapping to different data layouts. Let's take a look at the bit address representation of the absolute addresses for remaps to a cyclic layout and back to a blocked layout:

- For a blocked data layout the *first* $\lg P$ bits of the node's relative address will be the same as in the absolute address (the *processor's part* of the node's relative address). The *last* $\lg n$ bits will represent the local address of the node on the host processor.[1]

- For a cyclic data layout the *last* $\lg P$ bits of the absolute address will be the same for all the nodes that go on the same processor (the processor part). The *first* $\lg n$ bits of the absolute address represent the local address of the node on the host processor.

The conversion from the absolute address to the relative address, and vice-versa, in the case of a blocked to cyclic remap is suggested by Figure 3.2. Therefore, by knowing the absolute address of a node and the bit pattern representation of its relative address for the new data layout, we can easily determine what nodes get remapped on a specific processor.

We will now show why $N \geq P^2$ is required for the cyclic-blocked remapping strategy to work. Assume that $N < P^2$ and since $N = nP$, we get $\lg n < \lg P$. Consider the last stage of the bitonic sorting network. We know that the first $\lg P$ steps of this stage will execute locally under a cyclic layout. Since $\lg n + 1 \leq \lg P$, consider the first $(\lg n + 1)$ steps which will be part of the cyclic remap phase (which has $\lg P$ steps); each step performs a comparison between elements whose absolute addresses differ in only one bit. By the way a relative address is derived from the

[1]Observe that if we remap the nodes in the same order as in the initial blocked data layout the relative address is the same as the absolute address. However, for optimization purposes of the local computation, it may be necessary that we remap nodes on different local addresses as we will show in Chapter 4.

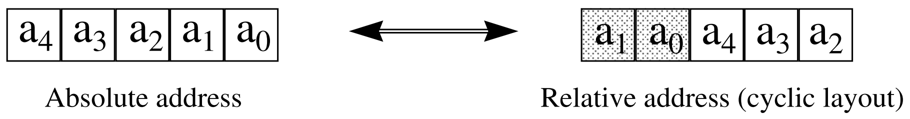


Figure 3.2: *Conversion from absolute to relative address for a remap to a cyclic layout (for $N = 32$ nodes and $P = 4$ processors - shaded areas represent the processor part of the address).*

absolute address for a cyclic layout results that also the relative addresses differ in only one bit. Furthermore the rank of this bit is different for each step. Therefore we will change at least $(\lg n + 1)$ bits in the relative address which is $\lg n + \lg P$ bits long; this is a contradiction with the fact that $\lg P$ bits of the relative address of the nodes remapped on the same processor are identical. Thus we need to have $\lg n \geq \lg P$, i.e. $N \geq P^2$.

Let's take a closer look at the butterfly network (Figure 2.4) where nodes are represented by their bit address representation. At each merging step we compare nodes whose absolute addresses differ in *only one bit*. More precisely at step s we compare nodes whose absolute addresses differ only at the s-th bit (bit 1 is the least significant bit; remember that we count steps backwards, i.e., the stage s starts with step s and ends with step 1). The following observation leads us to a communication optimal data layout (with respect to the number of data remaps) which will be derived in the next section:

Lemma 1 *After the first $\lg n$ stages (which can be entirely executed locally under a blocked layout) the maximum number of successive steps of the bitonic sorting network that can be executed locally, under any data layout, is $\lg n$ (where $n = N/P$, N=data size, P=number of processors).*

> **Proof:** The proof is straightforward. Assume that $\lg n + 1$ steps can be executed locally. Let's consider the relative address of a node at stage $\lg n + k$, where $0 < k \leq \lg P$. Assume that starting with step s, $0 < s \leq \lg n + k$, we can have more than $\lg n$ steps that execute locally.

We consider two cases:

- $s \geq \lg n + 1$. Consider the steps from s to $s - \lg n - 1$, within the stage $\lg n + k$. The absolute addresses of the node's compare pairs will change in a *different* bit at each step, therefore we will change at least $\lg n + 1$ bits out of the $\lg N$ bits of the absolute address.
- $s \leq \lg n$. Similarly, we'll change bits $s \ldots 1$ (within stage $\lg n + k$) and $\lg n + k + 1 \ldots \lg n + k + 1 - s$ (within stage $\lg n + k + 1$) in the node's absolute address pattern. Therefore we will change at least $\lg n + 1$ bits in the node's absolute address.

In both of the above cases $\lg n + 1$ bits in the absolute address are changed. Without loss of generality we can assume that these are the least significant $\lg n + 1$ bits of the address.

Note that if A is the absolute address of a node mapped on the processor then $A \oplus 2^i$, where $0 \leq i \leq \lg n$, is also an absolute address of a node on the same processor. Thus all addresses of the form $A \oplus i_0 2^0 \oplus i_1 2^1 \oplus \ldots \oplus i_{\lg n} 2^{\lg n} = A \oplus (i_0 2^0 + i_1 2^1 + \ldots + i_{\lg n} 2^{\lg n})$, where $i_k \in \{0, 1\}, k = \overline{0, \lg n}$.

Therefore, the total number of different addresses generated is $2^{\lg n + 1} = 2n$. This is a contradiction with the fact that we have n elements on each processor.

Therefore, $\lg n$ is the maximum number of successive steps of the bitonic sorting network (after the first $\lg n$ stages) that can be executed completely local. $\square$

The above lemma shows that there is a fundamental limitation on how much the communication part of the bitonic sort algorithm can be optimized. In other words there is a lower bound on the number of data remaps. In the following sections we derive a new data layout that will allow us to achieve this lower bound.

3.2 Deriving the Optimal Data Layout

The previous lemma tells us that, in order to avoid remote accesses, after at most $\lg n$ steps we have to remap the data again. A remap strategy that will execute exactly $\lg n$ steps locally before remapping again will generate the smallest possible number of remaps. Therefore, with respect to the number of data remaps, we will have a communication optimal parallel bitonic sort algorithm.

The example in Figure 3.3 ($N = 256$ elements, $P = 16$ processors) shows how we can remap the data so that after each remap operation exactly $\lg n$ steps will execute locally. The example shows only the last $\lg P$ stages of a bitonic sorting network and assumes that the first $\lg n$ stages execute locally under a blocked layout. In this example we execute only 7 data remaps, while in the cyclic-blocked implementation we were executing 8 remaps. Furthermore, we will show that at each remap we transfer less elements than in the case of a cyclic-blocked remap. Figure 3.4 presents the bit patterns of the absolute addresses of the nodes that go on the same processor at each remap operation in Figure 3.3, where the shaded bits represent the processor number. The bit pattern address representation will play an important role in deriving an optimal data layout.

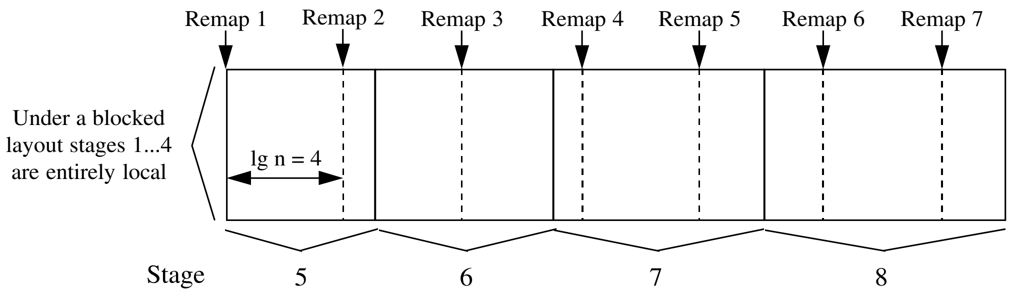


Figure 3.3: *Executing* $\lg n$ *steps locally after each remap operation (*$N = 256$ *elements,* $P = 16$ *processors,* $n = N/P = 4$ *elements per processor).*

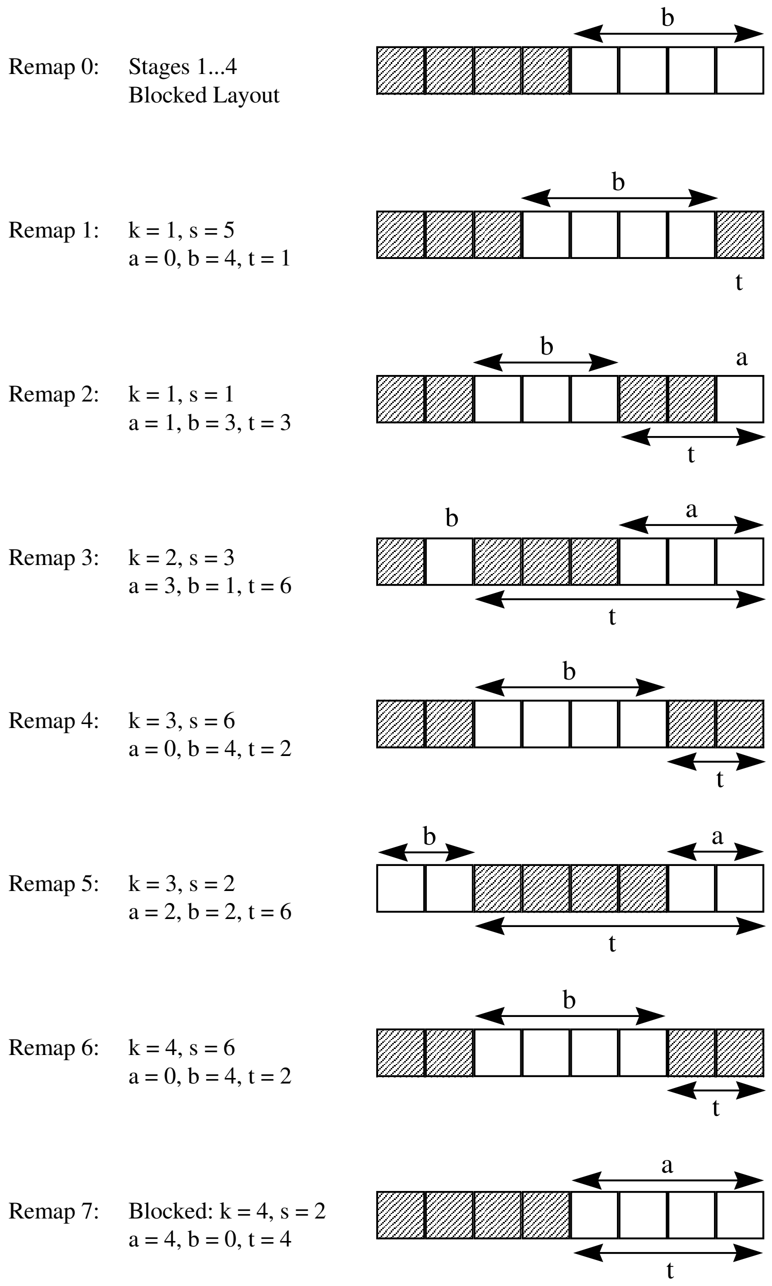


Figure 3.4: *Computing the pattern of the absolute address for each remap operation in Figure 3.3. The shaded bits represent the processor part of the address while the unshaded bits represent the local address part. The parameters k, s, a, b, t are introduced by Definition 7.*

Given the tuple $(stage, step)$, which uniquely identifies a column of the bitonic sorting network, our goal is to remap the elements at this specific step in such a way that the next $\lg n$ steps of the bitonic sorting network are executed locally. The essential observation is that in order to execute $\lg n$ steps locally the absolute addresses of the elements on a processor should match a specific bit pattern. As seen in the proof of Lemma 1 we generate all the possible n absolute addresses of the nodes that remap on the same processor by changing only $\lg n$ bits. We consider two cases depending whether we change or not the stage during the $\lg n$ steps following the remap operation. The first case is illustrated in Figure 3.5 where we have an *inside remap* $(s \geq \lg n)$. Here all the steps following the remap are inside the stage where the remap occurred and the bits that change in the absolute address are $s \ldots s - \lg n + 1$. All the other bits are unchanged and therefore we get the pattern for the absolute address presented in the figure. In the case when we change the stage (see Figure 3.6) we have a *crossing remap* $(s < \lg n)$; since we execute the last s steps of stage $\lg n + k$ and the first $\lg n - s$ steps of stage $\lg n + k + 1$ we will change bits $s \ldots 1$ and $\lg n + k + 1 \ldots k + s + 2$ and we get again a pattern for the absolute addresses of the nodes that remap on the same processor.

Since the shaded part in the absolute address is identical for all the elements on a processor we can consider this as being the processor number.

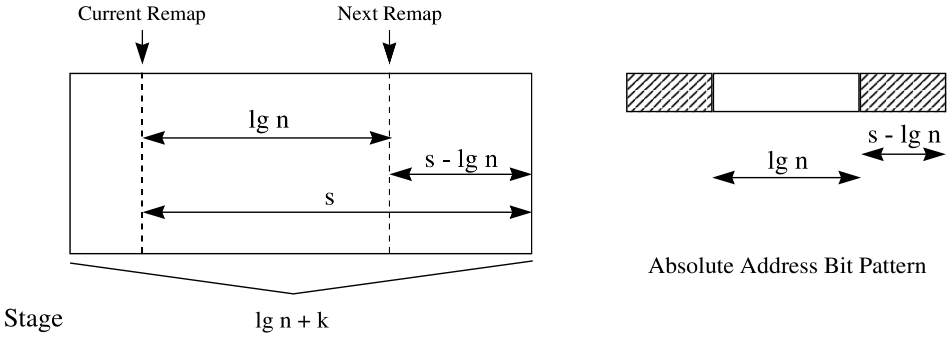


Figure 3.5: *Inside Remap and the corresponding absolute address bit pattern. The shaded bits represent the processor part of the address.*

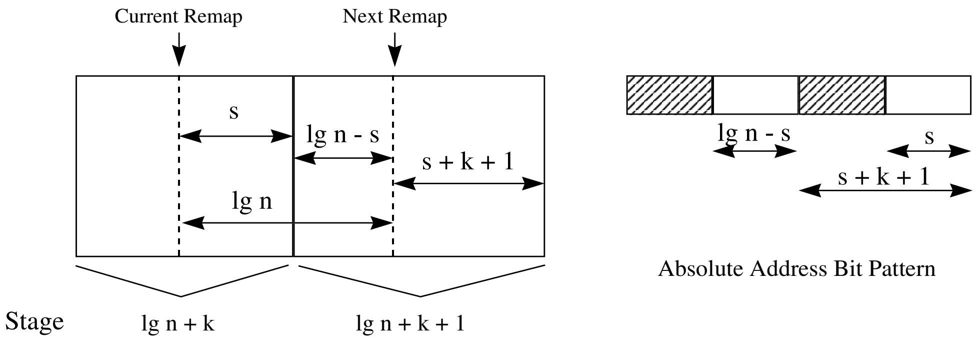


Figure 3.6: *Crossing Remap and the corresponding absolute address bit pattern. The shaded bits represent the processor part of the address.*

The parameters used to compute the relative address at each remap and the data layout used in the previous examples are introduced by the following definition:

Definition 7 (Smart Layout) *Given the tuple (stage* $= \lg n + k$*, step* $= s$*), where* $0 < k \leq \lg P$ *and* $0 < s \leq \lg n + k$ *(*N *= data size,* P *= number of processors,* $n = N/P$ *= elements per processor), we can define a* smart data layout *at step* s *within the stage* $\lg n + k$ *as the 5-tuple* (k, s, a, b, t) *where* a*,* b *and* t *are defined as follows:*

$$a = \begin{cases} 0 & s \geq \lg n \\ s & s < \lg n \end{cases}$$

$$b = \lg n - a$$

$$t = \begin{cases} s - \lg n & s \geq \lg n \\ s + k + 1 & s < \lg n \end{cases}$$

The above formulas change in the case $k = \lg P$ *and* $s \leq \lg n$ *(the last remap, when we remap to a blocked layout) to:*

$$\begin{aligned} a &= \lg n, \\ b &= 0, \\ t &= \lg n. \end{aligned}$$

Given the absolute address of a node, the relative address where the node is remapped is computed as presented in Figure 3.7 (for $s \geq \lg n$*) and Figure 3.8 (for* $s < \lg n$*).*

The meaning of the parameters a, b, t, their computing formulas and the translation from an absolute to a relative address for the smart data layout is suggested by Figure 3.7 and Figure 3.8, where a, b and t are expressed in steps of the bitonic sorting network. To find the processor number where the node is remapped we simply consider the shaded parts labeled A and C of the absolute address.

As specified in its definition the smart data layout depends on the current stage and step in the bitonic sorting network. Therefore we should keep track on how these parameters change when we perform a remap to a smart data layout execute $\lg n$ steps locally and then remap to another smart data layout. The generating formulas are straightforward; the stage where we will execute the next remap will be:

$$NextStage = \begin{cases} \lg n + k & s > \lg n \\ \lg n + k + 1 & s \leq \lg n \end{cases}$$

The next step where we execute the remap (within the stage previously computed) will be:

$$NextStep = \begin{cases} \lg n + k + 1 & t = 0 \\ t & t \neq 0 \end{cases}$$

The initial remap parameters (corresponding to the blocked data layout preceding the first smart remap) are:

$$\begin{aligned} k &= 0, \\ s &= \lg n + 1. \end{aligned}$$

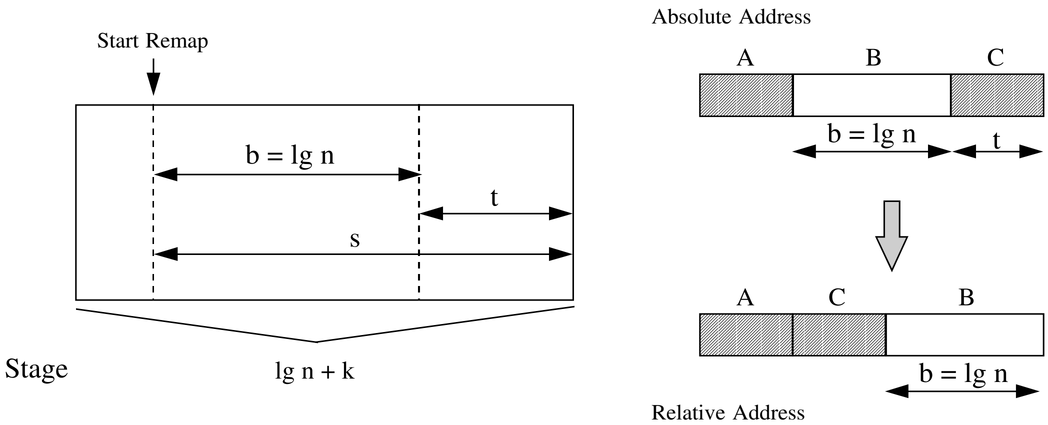


Figure 3.7: *An* Inside Remap *($s \geq \lg n$) for the Smart data layout.*

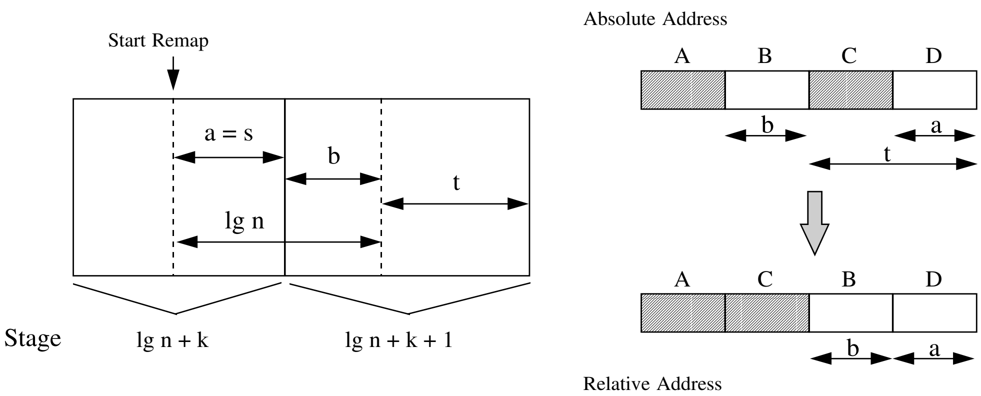


Figure 3.8: *A* Crossing Remap *($s < \lg n$) for the Smart data layout.*

The following lemma summarizes our observations:

Lemma 2 *The smart layout remaps the data such that it can execute exactly* $\lg n$ *steps locally.*

Proof: The proof is straightforward if we use the absolute and relative addresses of the node. We use the address translation from Figures 3.7 and 3.8 which shows the pattern of the absolute address of the nodes that go on the same processor.

We consider 2 cases:

- *Inside Remap* (Figure 3.7): $s \geq \lg n$. Executing b steps of the bitonic sorting network, starting with step s of stage $\lg n + k$, will change only the bits from $b + t$ to $t + 1$ in the absolute address representation of a node. As it results from Figure 3.7 and the definition of the smart remap nodes that change bits $b + t \ldots t + 1$ of the absolute address will reside on the same processor. Since $b = \lg n$ we can execute $\lg n$ steps locally.
- *Crossing Remap* (Figure 3.8): $s < \lg n$. In this case, executing $\lg n$ steps (a steps in stage $\lg n + k$ and b steps in stage $\lg n + k + 1$) is equivalent with changing the bits $b + t \ldots t + 1$ and $a \ldots 1$ in the absolute address representation of a node. As it results from the definition and the address translation (Figure 3.8), since $b + a = \lg n$, we'll have all the nodes that respect the above condition mapped on the same processor.

In both cases we can execute $\lg n$ steps of the bitonic sorting network completely locally. □

In the following we present our sorting algorithm based on the bitonic sorting network:

Algorithm 1 (Smart Layout Parallel Bitonic Sort) *The parallel bitonic sort algorithm for sorting N elements on P processors ($n = N/P$ elements per processor) starts with a blocked data layout and executes the first* $\lg n$ *stages entirely local. For the last lgP stages it periodically remaps to a smart data layout and executes* $\lg n$ *steps before remapping again.*

Clearly, the smallest number of communication steps is achieved if we use a remapping strategy that performs the smallest number of data remaps. Assuming that we don't replicate the data then, as we showed previously, for the last $\lg P$ stages of the bitonic sorting network $\lg n$ is the maximum number of steps that can be executed locally. The following theorem summarizes our observations:

Theorem 1 *Algorithm 1 uses the smallest possible number of data remaps.*

> **Proof:** In Algorithm 1 communication occurs only after the first $\lg n$ stages. Lemma 1 proves that $\lg n$ is the maximum number of steps of the bitonic sorting network that can be executed locally. Lemma 2 shows that under the smart data layout used by Algorithm 1 exactly $\lg n$ steps are executed locally. Thus, we perform the smallest possible number of remaps. □

It is also worth to point out that the new remapping technique does not impose any restriction on N and P (i.e. it is not necessary that $N \geq P^2$ like in the case of cyclic-blocked remap).

3.2.1 Communication Complexity

In this subsection we analyze the communication requirements of the new remapping strategy (which we call the *smart remap*) and compare it with the *cyclic-blocked remap*. By R_{Remap} we will denote the number of data remaps executed during the execution of the bitonic sort algorithm using the specified remapping strategy.

First, we point out that sorting using the cyclic-blocked remap requires:

$$R_{CyclicBlocked} = 2\lg P$$

remap operations (two remaps for each one of the last $\lg P$ stages). The smart remap strategy, however, performs a remap every $\lg n$ steps, therefore the total number of remaps is:

$$R_{Smart} = \lceil \lg P + \frac{\lg P(\lg P + 1)}{2\lg n} \rceil.$$

After each smart remap we will execute $\lg n$ steps locally except for the last one when we execute only

$$[\lg n \lg P + \frac{\lg P(\lg P + 1)}{2}] \bmod \lg n = \frac{\lg P(\lg P + 1)}{2} \bmod \lg n$$

steps. Observe that for $\frac{\lg P(\lg P+1)}{2} \leq \lg n$ we get $R_{Smart} = \lg P + 1$ and after the last remap we execute exactly $\frac{\lg P(\lg P+1)}{2}$ steps locally. For usual computations $N \gg P$, therefore in this case we perform $\lg P + 1$ remaps which is approximately *half* the number of remaps in the cyclic-blocked case. Another important characteristic of the smart remap is that it transfers less elements at each remap operation than the cyclic-blocked remap as we will show next.

The number of elements that a processor keeps can be determined analytically by analyzing the patterns of the absolute addresses of the nodes that reside on the processor before and after the remap is performed. The example in Figure 3.4 showed the patterns of the absolute addresses of nodes that were remapped on a processor at each smart remap. As noticed before, the shaded bits in the absolute address of a node represent the processor number where the node will be remapped; the non-shaded bits represent the local address of the node on the processor. Consider the bit pattern representations of the absolute address for two consecutive remaps (for example remaps 0 and 1 in Figure 3.4). The bits in the *local address* ($\lg n$ bits long) that become *shaded* are considered *changed*. In other words, elements that have the same *shaded* bits in the old local address as the corresponding bits in the processor part of the new address will be remapped on the same processor. Therefore, if we denote the number of changed bits considered before to be $N_{BitsChanged}$ then the number of elements that each processor will keep is

$$N_{keep} = 2^{\lg n - N_{BitsChanged}} = n/2^{N_{BitsChanged}}.$$

Furthermore, as we will prove next the processes will exchange elements in groups. The size of the group is $2^{N_{BitsChanged}}$ and each processor receives $2^{\lg n - N_{BitsChanged}}$ elements from any other processor in the group.

As we can see in Figure 3.4 from remap 0 to remap 1 only one bit changes, then we have in order 2, 3, 3, 4, 4, 2 bits that change for the subsequent remaps. Therefore we will transfer less elements than in the case of a cyclic-blocked remapping strategy where the number of bits that change is always $\lg P = 4$.

To find the number of bits of the absolute address that change at a smart remap we simply have to compare the bit patterns of the absolute addresses. The following two lemmas summarize our observations:

Lemma 3 *Given the tuple (k, s), where $0 < k \leq \lg P$, $0 < s \leq \lg n + k$ and $n \geq P$, a smart remap at step s within stage $\lg n + k$ will change exactly*

$$N_{BitsChanged} = \begin{cases} k & s \geq \lg n \\ k+1 & s < \lg n \end{cases}$$

bits in the absolute address pattern of nodes that remap on a processor.

For $n < P$ the formula changes only when $k > \lg n$ to

$$N_{BitsChanged} = \lg n.$$

In the special case $k = \lg P$ and $s \leq \lg n$ (the last remap) the formula changes to:

$$N_{BitsChangedLastRemap} = \begin{cases} s & s \leq \lg P \\ \lg P & s > \lg P \end{cases}$$

Proof: We will make again use of the bit pattern of the absolute addresses of the node.

We consider four cases:

- Current remap is a *crossing* remap, previous remap was an *inside* remap ($s < \lg n$). Figure 3.9 presents this case. As we can see $k+1$ bits are changed (*non-shaded* bits in the old address that become *shaded* in the new address).
- Current remap is a *crossing* remap, previous remap was a *crossing* remap ($s < \lg n$). Figure 3.10 presents this case. As before we get $N_{BitsChanged} = k + 1$.

- Current remap is an *inside* remap, previous remap was a *crossing* remap ($s \geq \lg n$). Figure 3.11 presents this case. By intersecting the two bit patterns the number of bits that change will be:

$$\begin{aligned} N_{BitsChanged} &= b + \min(s - \lg n, s - k) \\ &= b + s - \max(\lg n, k) \\ &= \lg n + k - \max(\lg n, k). \end{aligned}$$

  If $n \geq P$ we get $\lg n \geq \lg P \geq k$, therefore $N_{BitsChanged} = k$.
  If $n < P$ the formula changes only when $k > \lg n$ to $N_{BitsChanged} = \lg n$.

- Current remap is an *inside* remap, previous remap was an *inside* remap ($s \geq \lg n$). This case can only occur when $n < P$ and $k > \lg n$ and it is presented in Figure 3.13. As we can easily see the number of changed bits is always $N_{BitsChanged} = \lg n$.

The above four cases cover all possible situations for a remap from a smart data layout to another smart data layout. However, in the case when $k = \lg P$ and $s \leq \lg n$ we perform the last remap and we will execute less than $\lg n$ steps after the remap operation. Because the absolute address bit pattern after the remap changes to a blocked layout we treat this case separately.

We distinguish 2 possible situations:

- $s \leq \lg P$ - In this case the previous remap was an *inside* remap.
- $s > \lg P$ - In this case the previous remap was a *crossing* remap.

The bit patterns of the absolute addresses for these two cases are shown in Figure 3.12. Since $s \leq \lg n$, it is straightforward to check that the formulas for $N_{BitsChangedLastRemap}$ are correct. $\square$

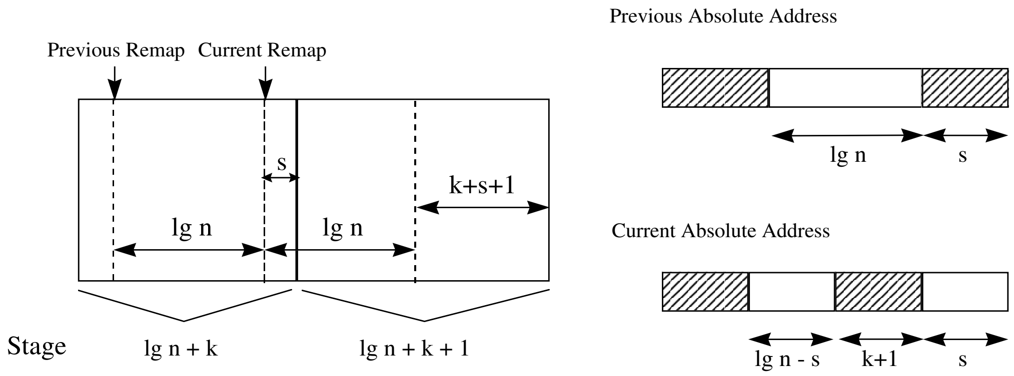


Figure 3.9: Current remap is *crossing* remap, previous remap was *inside* remap.

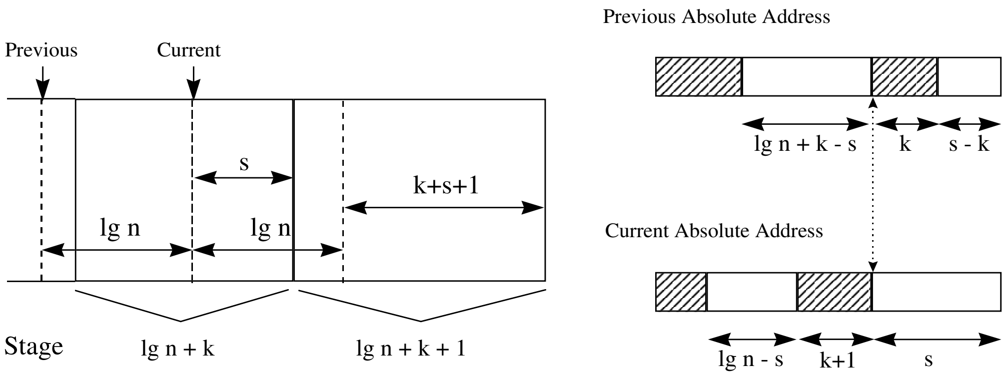


Figure 3.10: Current remap is *crossing* remap, previous remap was *crossing* remap.

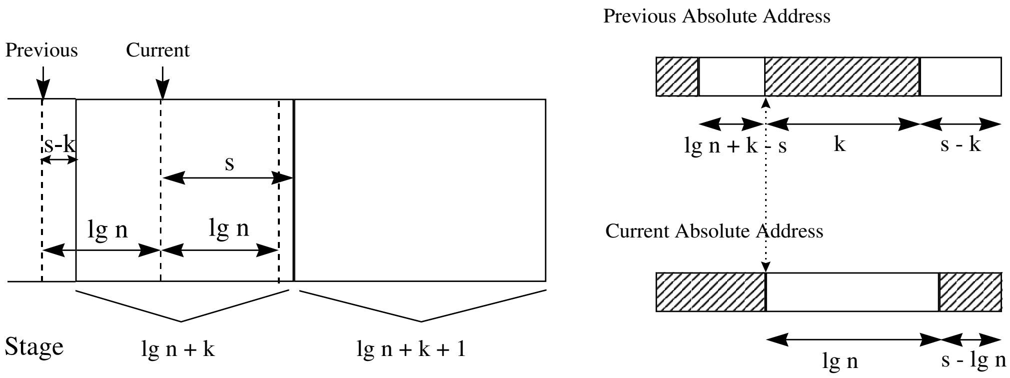


Figure 3.11: Current remap is *inside* remap, previous remap was *crossing* remap.

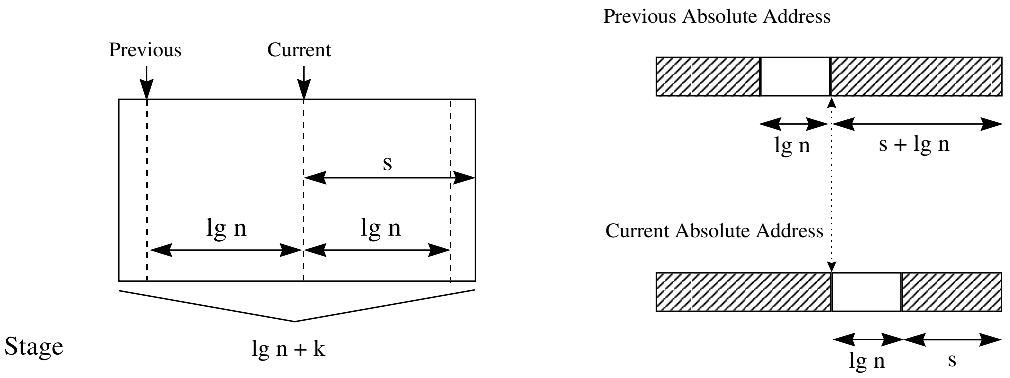
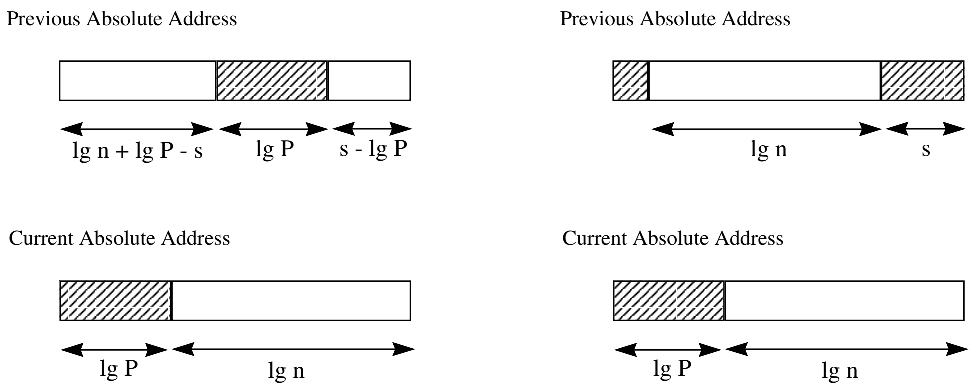


Figure 3.12: Current remap is *inside* remap, previous remap was *inside* remap.



a) $s \geq \lg P$ b) $s < \lg P$

Figure 3.13: *Possible bit patterns for the last remap.*

The interesting case and the one naturally occurring in practice is when $n \geq P$ and in the following we will consider only this case. As noted in the previous proof the cases $s \geq \lg n$ and $s < \lg n$ correspond to an *inside* and a *crossing* remap respectively, thus an equivalent formulation of the above formulas is that a smart remap within stage $\lg n + k$ will change exactly k bits of the absolute address (we consider a $\lg n$ steps smart remap to be part of stage $\lg n + k$ if it ends within the stage, i.e. the last step of the $\lg n$ steps following the remap is part of the stage).
The following lemma characterizes the communication complexity at a smart remap operation:

Lemma 4 *Consider a smart remap operation (introduced by Definition 7) and the number of bits that change ($N_{BitsChanged}$) for that remap (introduced by Lemma 3). Then the following take place:*

- *Processors communicate in groups of $2^{N_{BitsChanged}}$ processors. A given processor $PROC$ will communicate only with processors (having consecutive numbers) in the group starting with processor number $2^{N_{BitsChanged}} \lfloor PROC/2^{N_{BitsChanged}} \rfloor$.*
- *Each processor keeps $n/2^{N_{BitsChanged}}$ elements and sends $n/2^{N_{BitsChanged}}$ elements to every other processor in its group.*

Proof:

- Obviously when we remap an element the bit pattern of the processor part changes in $N_{BitsChanged}$ new positions and remains unchanged in the $\lg P - N_{BitsChanged}$ positions (see Figures 3.9– 3.13 used in the proof for the previous lemma). Also notice that the bits that change in the processor part are in consecutive locations. Since in the new positions we can have all possible combinations (these were previously part of the local address where we have all possible $\lg n$ combinations) we have $2^{N_{BitsChanged}}$ consecutive processors to communicate with.

  Thus, the total of P processors is divided in $2^{\lg P - N_{BitsChanged}}$ groups. Therefore the size of the group of processors that communicate is $2^{N_{BitsChanged}}$ and the number of the first processor in the group is $2^{N_{BitsChanged}} \lfloor PROC/2^{N_{BitsChanged}} \rfloor$.
- For each $N_{BitsChanged}$ bits that are modified in the bit pattern of the processor part we have $N_{BitsChanged}$ bits that are modified in the local address part. Elements whose absolute addresses are identical in these positions will be remapped on the same processor. Therefore, each processor in the group will get $2^{\lg n - N_{BitsChanged}}$ elements. Thus, each processor keeps $n/2^{N_{BitsChanged}}$ elements and sends $n/2^{N_{BitsChanged}}$ elements to every other processor in its group.

□

An immediate consequence of the previous lemma is that communication is well-balanced and contention is minimized.[2]

[2] Communication takes place between consecutive processors in a group (all-to-all communication

Now we can compute the total volume of elements transfered for the new algorithm and compare it with the volume of elements transfered in the cyclic-blocked implementation. By V_{Remap} we will denote the volume, i.e. the total number of elements per processor transfered during the execution of the bitonic sort algorithm using the specified remapping strategy. For the cyclic-blocked algorithm each processor exchanges, at each remap, $(P-1)\frac{n}{P}$ elements. Thus for the $2\lg P$ remaps performed by the cyclic-blocked strategy the total number of transfered elements per processor will be:

$$V_{CyclicBlocked} = 2n(1-\frac{1}{P})\lg P.$$

For the smart remapping strategy each processor transfers $n(1-1/2^{N_{BitsChanged}})$ at each remap. For all the $N_{SmartRemaps}$ phases the total number of transfered elements per processor will be:

$$V_{Smart} = n * \sum_{k=1}^{N_{SmartRemaps}} (1-\frac{1}{2^{N_{BitsChanged}[k]}}).$$

where $N_{BitsChanged}[k]$ is the number of changed bits at the k-th remap. To derive an approximation of the previous formula we need to find out where do smart remaps occur inside a stage. One important observation is that we can have at most one *crossing remap* per stage. If $n \geq P$, i.e., $\lg n + k \leq 2\lg n$ we can have at most two smart remaps per stage, therefore to derive the exact formula we need to find the stages that can have two remaps ending within that stage. This can be done by computing for each stage $\lg n + k$ the step s_k at which we change the data layout for the first time within that stage (i.e. where we perform the first remap within that stage):

Consider the general case from Figure 3.14. Observe that the remap shown will end within stage $\lg n + k$ if and only if $a_k < \lg n$. We will have the second remap

within the group), and the amount of transfered data is direct proportional with the size of the group. This is especially beneficial for network architectures like fat-trees because we avoid contention at the top switch-router of the fat-tree.

ending within stage $\lg n+k$ if and only if $s_k \geq \lg n$. Thus we get the following formula:

$$\begin{aligned} s_k &= \lg n + k - b_k \\ &= k + a_k \\ &= k + [\frac{k(k-1)}{2} \bmod \lg n]. \end{aligned}$$

With the previous observations the formula becomes:

$$s_k = \begin{cases} \lg n + k & a_k = 0 \\ k + a_k & a_k > 0 \end{cases}$$

where $a_k = k(k-1)/2 \bmod \lg n$. The case $a_k = 0$ means we start an *inside remap* right at the beginning of the stage and therefore we cannot have two remaps ending within that stage. In the case $a_k > 0$ we will have the next remap ending within the stage if and only if $s_k \geq \lg n$.

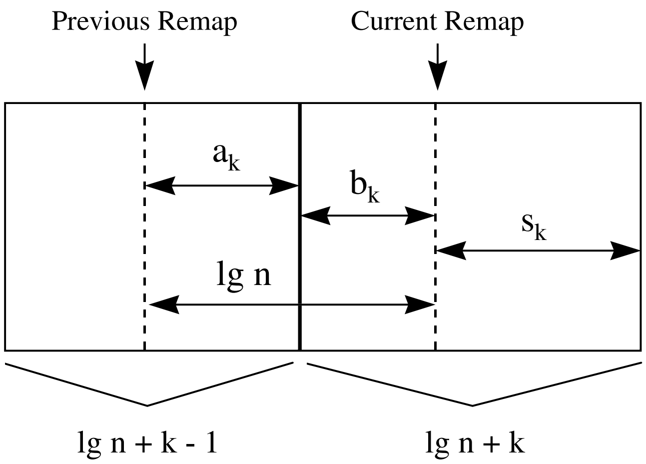


Figure 3.14: *Finding where we change the data layout for the first time inside a stage.*

Therefore if $\lg n + k > s_k \geq \lg n$ we have two remaps ending within stage $\lg n + k$ and in this case the number of bits changed will be k. Again, the last stage ($\lg n+\lg P$) becomes a special case because after the last smart remap the remaining steps will end within this stage. The number of bits changed at the last remap will be determined according to Lemma 3, therefore we should count the number of elements transfered at this remap separately. Now we can break down the types of remaps into three categories:

- *OutRemap* — remap for which $a_k < 0$ or $s_k = \lg n + k$. Observe that this includes all *crossing remaps* plus *inside remaps* that start exactly at the beginning of a stage. The important observation is that we have *exactly* one OutRemap per stage (i.e. exactly one OutRemap ending within each stage of the bitonic sorting network).
- *InRemap* — remap for which $\lg n + k > s_k \geq \lg n$. This includes all *inside remaps* that do not start exactly at the beginning of a stage. This guarantees that we will have two remaps ending within that stage.
- *LastRemap* — the last smart remap. Notice that after this remap we may execute less than $\lg n$ steps and therefore we consider it separately.

Thus, the exact formula for the total number of elements per processor transfered during the entire sorting process becomes:

$$\begin{aligned}
V_{Smart} &= V_{OutRemap} + V_{InRemap} + V_{LastRemap}. \\
V_{OutRemap} &= n * \sum_{k=1}^{\lg P} (1 - 1/2^k). \\
V_{InRemap} &= n * \sum_{\substack{k=1 \\ \lg n + k > s_k \geq \lg n}}^{\lg P} (1 - 1/2^k). \\
V_{LastRemap} &= n * (1 - 1/2^{N_{Last}}).
\end{aligned}$$

where N_{Last} represents the number of bits changed at the last remap operation.

After simplifying $V_{OutRemap}$ we get the general formula:

$$V_{Smart} = n(\lg P + 1/P - 1/2^{N_{Last}} + \sum_{\substack{k=1 \\ \lg n + k > s_k \geq \lg n}}^{\lg P} (1 - 1/2^k)).$$

We are interested to approximate the ratio of the number of transfered elements under the cyclic-blocked and the smart remapping strategies. Observe that for a fixed P, if $\frac{\lg P(\lg P+1)}{2} \leq \lg n$ then

$$\begin{aligned} R_{Smart} &= \lg P + 1, \\ s_k &< \lg n \end{aligned}$$

for any $1 \leq k \leq \lg P$. For this case the we don't have any $InRemaps$ ($V_{InRemap} = 0$) and because the number of steps after the last remap is $\frac{\lg P(\lg P+1)}{2} \geq \lg P$ we have

$$N_{Last} = \lg P.$$

Thus the formula becomes:

$$\begin{aligned} V_{Smart} &= n(\lg P + 1/P - 1/2^{\lg P}) \\ &= n \lg P. \end{aligned}$$

Since for usual computations $n \gg P$, the ratio will be:

$$V_{CyclicBlocked}/V_{Smart} \simeq 2(1 - 1/P).$$

This means that we transfer approximately $2(1 - 1/P)$ times less elements than for the cyclic-blocked remap.

3.2.2 Optimal Remapping Strategy

In the previous subsection we estimated the communication complexity of the smart remap bitonic sort algorithm by deriving the total volume of transfered data per processor. Remember that the algorithm starts with a blocked data layout and executes the first $\lg n$ stages entirely local. For the last $\lg P$ stages it periodically remaps to a smart data layout and executes $\lg n$ steps before remapping again. It may be the case that after the last remap we are left with less that $\lg n$ steps.

This situation raises the following question: What if we *shift* the remaps to the left or right so that we execute a different number of steps after the last remap? Will that change the total number of elements transfered? The answer is *YES* and the optimal remapping strategy (with respect to the total number of elements transfered) is given by the following Lemma:

Lemma 5 *Let* $N_{RemainingSteps} = \frac{\lg P(\lg P+1)}{2} \bmod \lg n$. *Consider the following remapping strategies for the last* $\lg P$ *stages of the bitonic sorting network:*

- HeadRemap — *Execute locally* $\lg n$ *steps after each remap except the* last *one when we execute* $N_{RemainingSteps}$ *steps.*

- TailRemap — *Execute locally* $\lg n$ *steps after each remap except the* first *one when we execute* $N_{RemainingSteps}$ *steps.*

- MiddleRemap1 — *Execute locally* $\lg n$ *steps after each remap except the* first *and the* last *one when we execute respectively* $N_{StepsHead}$ *and* $N_{StepsTail}$ *steps, where* $N_{StepsHead} + N_{StepsTail} = N_{RemainingSteps}$ *and* $N_{StepsHead} > 0$, $N_{StepsTail} > 0$ *(this corresponds to shifting the remaps to the right by* $N_{StepsHead}$ *steps; also observe that the number of remap operations has increased by 1).*

- MiddleRemap2 — *Execute locally* $\lg n$ *steps after each remap except the* first *and the* last *one when we execute respectively* $N_{StepsHead}$ *and* $N_{StepsTail}$ *steps, where* $N_{StepsHead} + N_{StepsTail} = \lg n + N_{RemainingSteps}$ *and* $N_{StepsHead} > 0$,

> $N_{StepsTail} \geq N_{RemainingSteps}$ *(this corresponds to shifting the remaps to the left by* $\lg n - N_{StepsHead}$ *steps; also observe that the number of remap operations is unchanged).*

Denote with $V_{HeadRemap}$*,* $V_{TailRemap}$*,* $V_{MiddleRemap1}$*,* $V_{MiddleRemap1}$ *respectively the total number of elements transfered for the above remapping strategies. Then for* $n \geq P^2$ *the following relationships hold:*

$$\begin{aligned} V_{TailRemap} &\leq V_{HeadRemap}, \\ V_{HeadRemap} &< V_{MiddleRemap1}, \\ V_{TailRemap} &\leq V_{MiddleRemap2}. \end{aligned}$$

For $\frac{\lg P(\lg P+1)}{2} \leq \lg n$ *we will have:*

$$V_{HeadRemap} = V_{TailRemap}.$$

Proof:

We will prove the inequalities separately:

- $V_{HeadRemap} < V_{MiddleRemap1}$ — In the previous subsection we have already computed the formula for $V_{HeadRemap}$. Observe that for a $MiddleRemap1$ we introduce a new remap operation at the beginning of stage $\lg n + 1$ at which we will transfer $N/2$ elements. The number of elements transfered at the following remaps (except the last one) will be greater or equal that before because we may have remaps ending in the next stage than before. Consider that the number of bits that change at the last remap for the $HeadRemap$ was N_{BC}. Then for the $MiddleRemap1$ the number of bits that change at the last remap N_{BC1} will satisfy:

  $$1 \leq N_{BC1} \leq N_{BC}.$$

  Therefore:

  $$\begin{aligned} V_{MiddleRemap1} - V_{HeadRemap} &\geq n/2 + n(1 - 1/2^{N_{BC1}}) - n(1 - 1/2^{N_{BC}}) \\ &\geq n(1/2 - 1/2^{N_{BC1}}) \\ &\geq 0. \end{aligned}$$

- $V_{TailRemap} \leq V_{MiddleRemap2}$ — Consider the formula for $V_{MiddleRemap2}$:

$$\begin{aligned} V_{MiddleRemap2} &= V_{OutRemap} + V_{InRemap} + V_{LastRemap} \\ &= V_{OutRemap} + V_{InRemap} + n(1 - 1/2^{N_{BitsChanged}}). \end{aligned}$$

where $N_{BitsChanged}$ is the number of bits changed at the last remap and is given by Lemma 3.

TailRemap is obtained by shifting the *MiddleRemap*2 to the left by $\lg n - N_{StepsTail}$ steps. This will have the effect of increasing the number of elements transfered at the last remap to $n(1 - 1/2^{\lg P})$. We may also have remaps moving to a lower stage thus transferring less elements for these remaps. Observe that *InRemaps* cannot move to a lower stage, therefore $V_{InRemap}$ remains unchanged. A remap will move to a lower stage (i.e. change the number of bits from k to $k-1$) if (see Figure 3.15):

$$a_k \geq N_{StepsTail}.$$

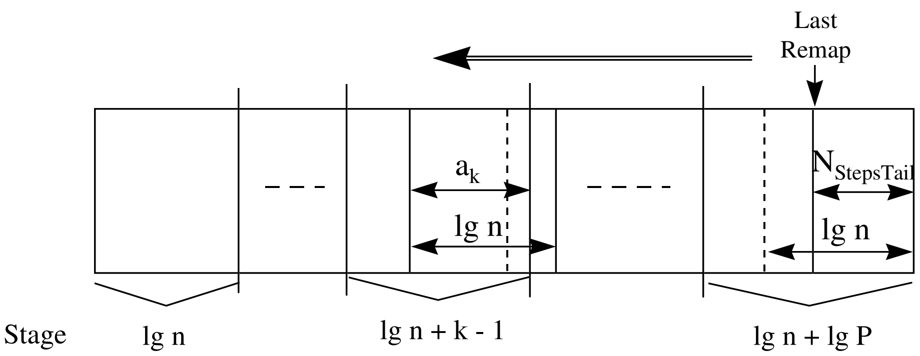


Figure 3.15: *Shifting the remaps to the left can decrease the volume of transfered data.*

We get:

$$V_{MiddleRemap2} - V_{TailRemap} =$$

$$= n\left(\sum_{\substack{k=1 \\ a_k \geq N_{StepsTail}}}^{\lg P} 1/2^{k-1}\right) + n(1 - 1/2^{N_{BitsChanged}}) - n(1 - 1/2^{\lg P})$$

$$= n\left(\sum_{\substack{k=1 \\ a_k \geq N_{StepsTail}}}^{\lg P} (1/2^{k-1}) - 1/2^{N_{BitsChanged}} + 1/2^{\lg P}\right).$$

If $N_{StepsTail} \geq \lg P$ then $N_{BitsChanged} = \lg P$ and therefore

$$V_{MiddleRemap2} - V_{TailRemap} \geq 0.$$

If $N_{StepsTail} < \lg P$ then $N_{BitsChanged} = N_{StepsTail}$. We can have one of the following situations:

– $a_{N_{StepsTail}} \geq N_{StepsTail}$. Then

$$\begin{aligned} V_{MiddleRemap2} - V_{TailRemap} &\geq n(1/2^{N_{StepsTail}-1} - 1/2^{N_{StepsTail}} + 1/2^{\lg P}) \\ &\geq 0. \end{aligned}$$

– $a_{N_{StepsTail}} < N_{StepsTail}$. The recurrence relation for a_k is $a_{k+1} = (a_k + k) \bmod \lg n$. Then $a_{N_{StepsTail}+1} = (a_{N_{StepsTail}} + N_{StepsTail}) \bmod \lg n$. Since $\lg n \geq 2\lg P$ $a_{N_{StepsTail}} + N_{StepsTail} < 2\lg P \leq \lg n$. Therefore $a_{N_{StepsTail}+1} = a_{N_{StepsTail}} + N_{StepsTail} \geq N_{StepsTail}$. Thus:

$$\begin{aligned} V_{MiddleRemap2} - V_{TailRemap} &\geq n(1/2^{N_{StepsTail}} - 1/2^{N_{StepsTail}} + 1/2^{\lg P}) \\ &\geq 0. \end{aligned}$$

- $V_{TailRemap} \leq V_{HeadRemap}$ — This results as a particular case of the previous proof. For $N_{StepsTail} = N_{RemainingSteps}$ a $MiddleRemap2$ becomes a $HeadRemap$ and the rest follows.

As noticed before for $\frac{\lg P(\lg P+1)}{2} \leq \lg n$ we have:

$$\begin{aligned} N_{RemainingSteps} &= \frac{\lg P(\lg P+1)}{2}, \\ N_{BitsChangedLastRemap} &= \lg P. \end{aligned}$$

Since $a_{k+1} = (a_k + k) \bmod \lg n$ from the above results:

$$a_k < N_{RemainingSteps},$$

for any $1 \leq k \leq P$. Therefore:

$$V_{TailRemap} = V_{HeadRemap} = V_{MiddleRemap2} = n \lg P.$$

□

The above lemma shows that the total volume of elements transfered can be influenced by the way we execute the smart layouts. For usual computations however (i.e. $\frac{\lg P(\lg P+1)}{2} \leq \lg n$) the total volume doesn't change if we execute either a $HeadRemap$

or a *TailRemap*. Another interesting question is "What is the minimum number of elements that are transfered during a remap-based bitonic sort?". We believe that the *TailRemap* presented above achieves this lower bound, however this was beyond the scope of this thesis.

3.3 Remap Implementation

In this section we present an efficient implementation technique for remapping the data. In the previous sections we have shown how, given the bit pattern of the absolute addresses before and after the remap, we can compute, for each element, its next relative address.

The problem that arises now is how can we transfer all these elements to their new locations as fast as possible. The straightforward solution would be to send one element at a time to its corresponding location. This is what we call a short message transfer, because it transfers only a small amount of data in a message (in our case we transfer four bytes for an integer). Modern parallel machines, however, have support for long message transfers, i.e., pack more elements that go on a processor in a single message and perform just one communication step (a long message transfer). Thus we can take advantage of the full bandwidth of the network (see [AISS95, SS95]) by grouping elements into long messages. In the following we will present a general method for remapping the data between two different data layouts using long messages based on the bit pattern of the absolute address for the respective data layouts.

Consider the two data layouts (represented by absolute address bit patterns) from Figure 3.16. The goal is to remap the elements from $Layout_0$ to $Layout_1$ using long messages. This operation will involve three steps (see Figure 3.17):

- "packing" the elements that go on the same processor in a single long message,
- sending each long message to its destination processor,
- "unpacking" the elements from the long message to their location on the desti-

nation processor.

Observe that the "packing" and "unpacking" operations are not required in the short message transfer, however while the overhead they introduce is a substantial fraction compared to the actual long message transfer time it is much smaller than the short message transfer time and results in a huge overall gain as we will show in Chapter 5.

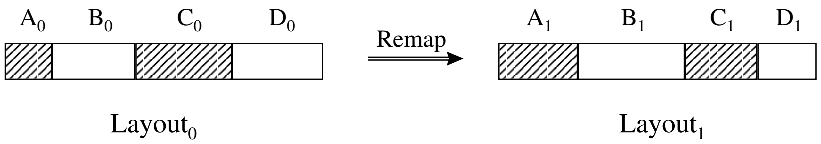


Figure 3.16: *Remap between two different data layouts.*

3.3.1 Data Packing and Unpacking

Consider the two bit patterns from Figure 3.16. Figure 3.18 presents the *pack mask* that we used for the "packing" operation. The purpose of this mask is to compute the new processor number and the new local address for each element in $Layout_0$. The mask is $\lg n$ bits long and was obtained by taking the local address in $Layout_0$ (parts marked B_0 and D_0) and "shading" the bits that become part of the processor number in $Layout_1$. Observe that the number of "shaded" bits is exactly $N_{BitsChanged}$ (which we denote by r in the figures) and represents the offset of the destination processor in its group (see Lemma 4). The "unshaded" fields are used to pack the data into the long message as suggested by Figure 3.20.

Observe that the entire operation can be implemented using three nested loops in which we transfer blocks of size 2^y. Elements that are remapped on the same processor will be transfered in a single long message of size $2^{\lg n - r}$.

The 2^r blocks of size $2^{\lg n-r}$ will be transfered as follows: the i-th block on processor j (j is the offset of the processor in its group) will go on the i-th processor in the group as the j-th block.

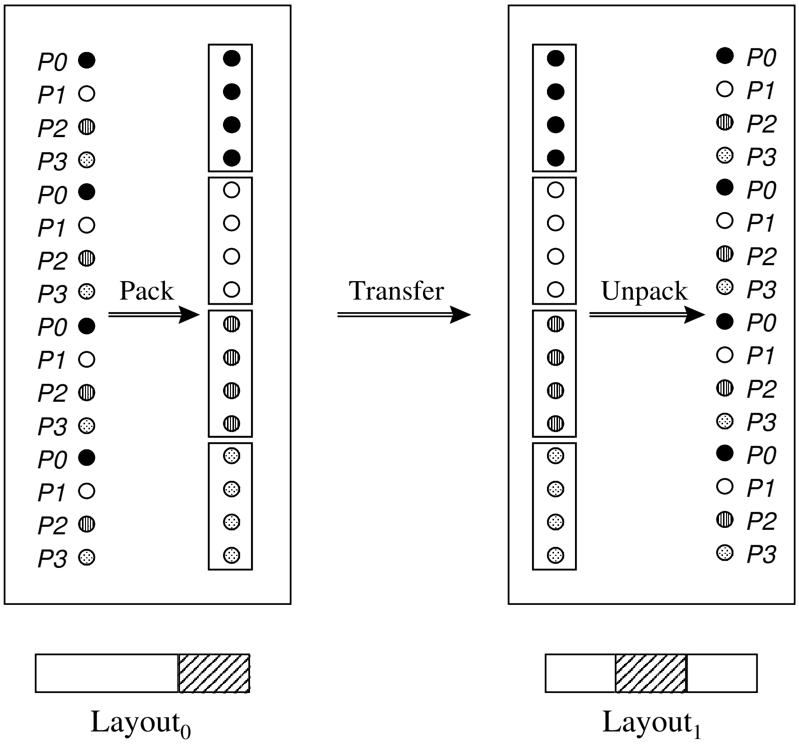


Figure 3.17: *A long message transfer consists of three different phases:* packing *of data into long messages, the actual* transfer *of the long messages and the* unpacking *of data from the long message. The figure presents a long message remap from $Layout_0$ to $Layout_1$. The data presented is on one processor. The labels next to the elements represent their destination processor (on the left) and the source processor (on the right).*

The unpack operation is the reverse of the pack operation. For this we need an *unpack mask* which is derived similarly, but now we exchange the roles of $Layout_0$ and $Layout_1$ and consider the fields that turn "shaded" in the local address of $Layout_1$. Figure 3.19 shows the construction of the unpack mask. Figure 3.21 presents how elements are extracted from the long message and assigned to their location on the destination processor.

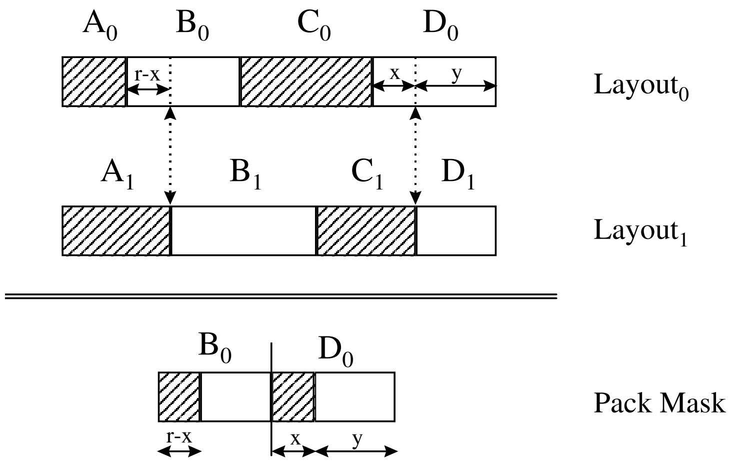


Figure 3.18: *Packing mask.*

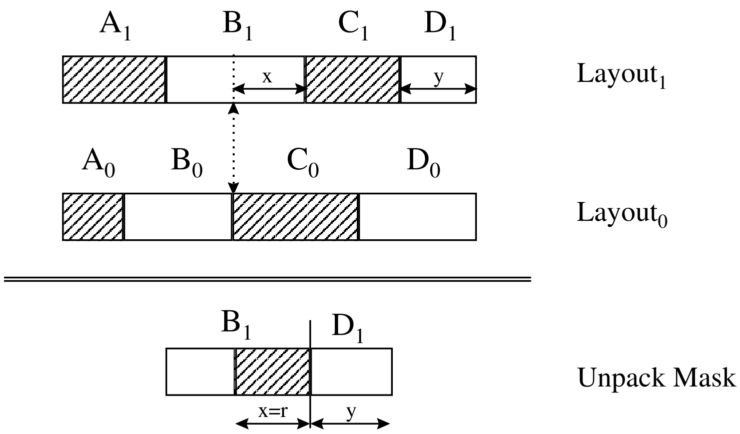


Figure 3.19: *Unpacking mask.*

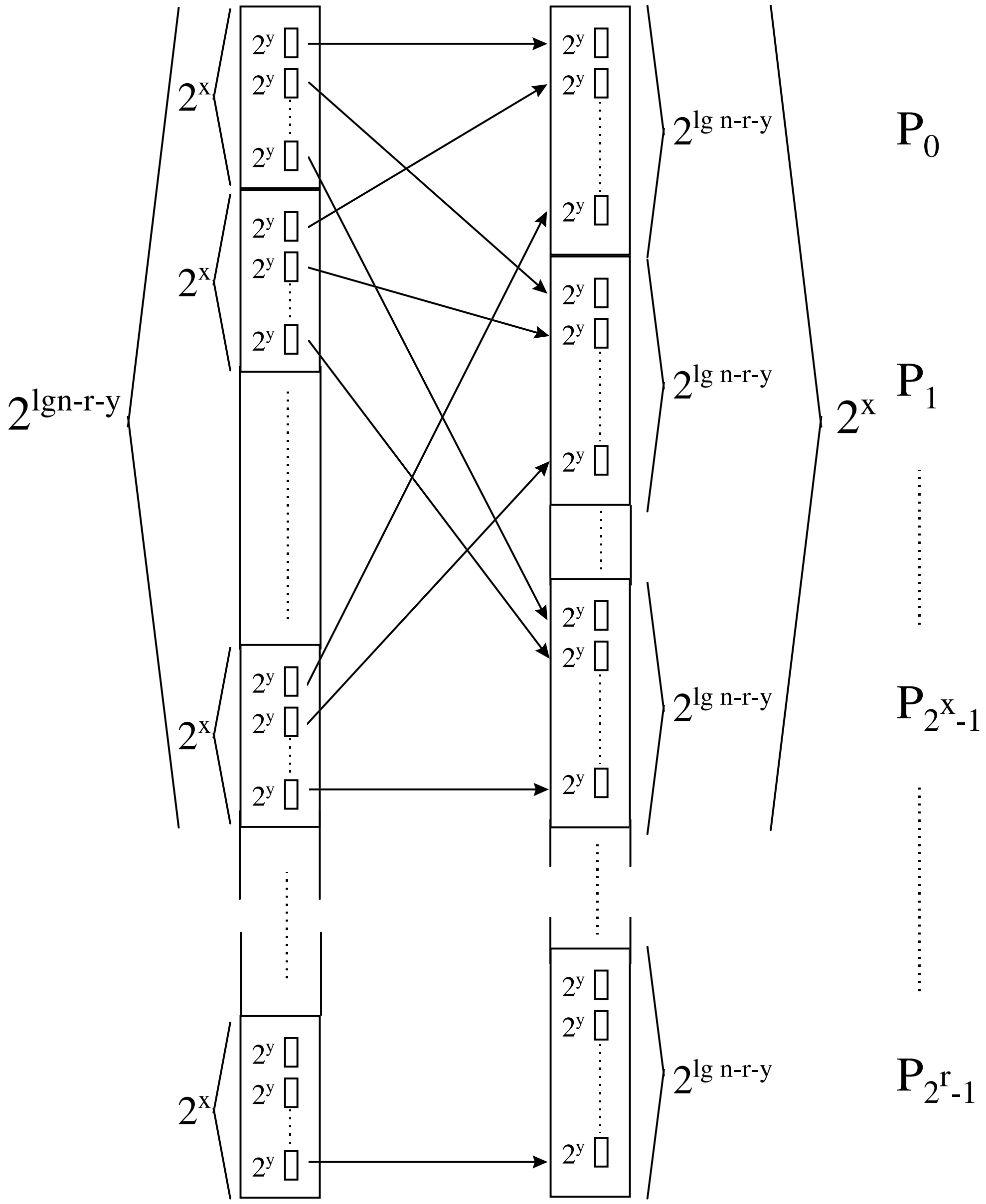


Figure 3.20: *Packing the data into long messages using the pack mask. The data presented is on one processor.*

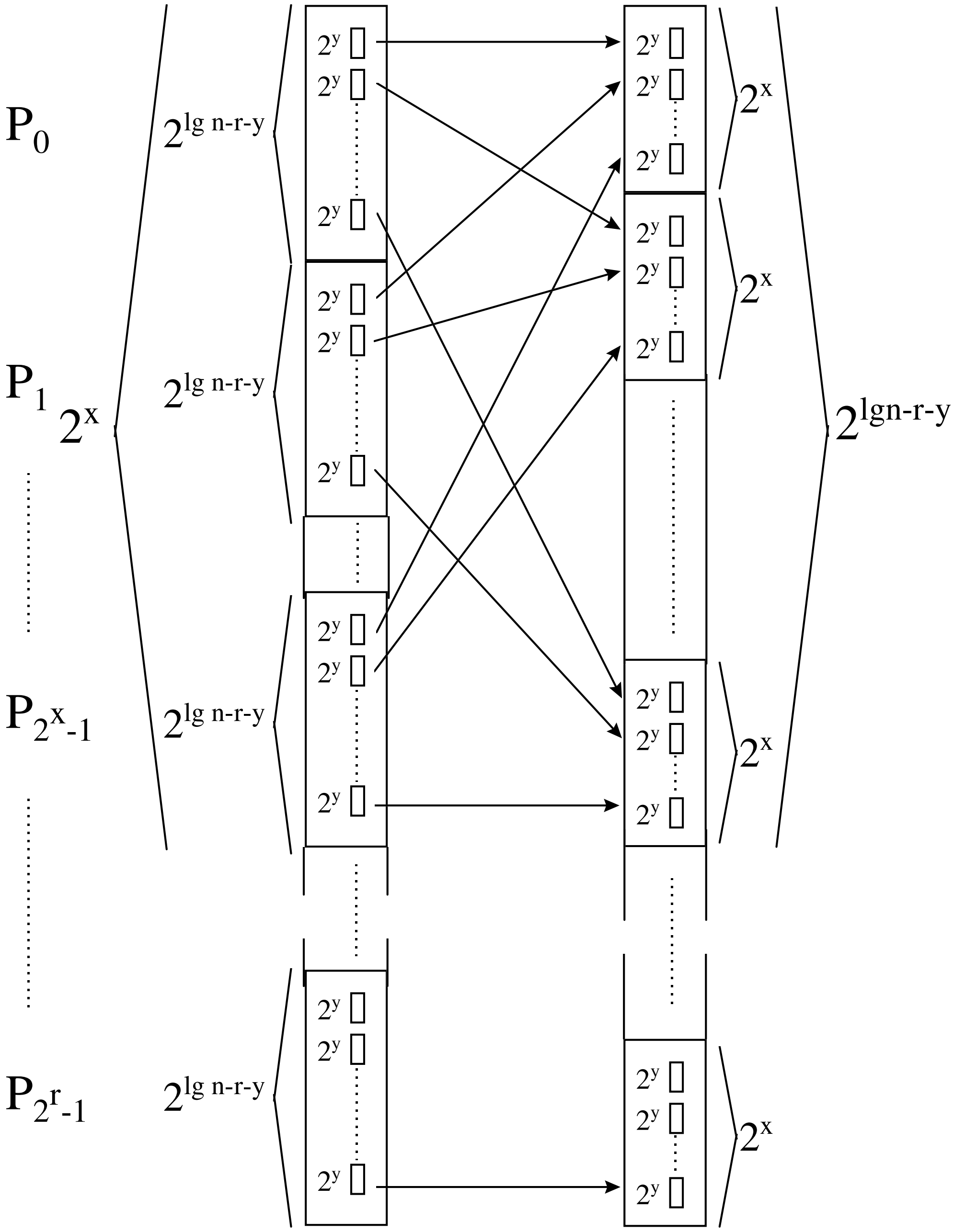


Figure 3.21: *Unpacking the data from the long messages using the unpack mask. The data presented is on one processor.*

3.4 Deriving a Communication Optimal Algorithm

In this section we analyze the communication in a remap based bitonic sort algorithm. As we have shown previously there is a data layout which minimizes the number of communication steps. However the total communication time of a parallel algorithm depends on other factors as well. Analyzing the algorithm under a "realistic" model of parallel computation which captures the existing overheads of modern hardware reveals that the important factors that influence the communication time are:

- total number of remaps
- total volume of elements transfered
- total number of messages transfered

For our study we have chosen the LogP [CKP+93] and LogGP [AISS95] models of parallel computation.

3.4.1 The LogP and LogGP models

The LogP model reflects the convergence of parallel machines towards systems formed by a collection of complete computers, each consisting of a microprocessor, cache and large DRAM memory, connected by a communication network [CKP+93].

Under the LogP model the processors communicate by point-to-point messages and communication performance is characterized by the following parameters:

- L: an upper bound on the *Latency*, incurred in sending a message from its source processor to its target processor.
- o: the *overhead*, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time the processor cannot perform other operations.

- g: the *gap* between messages, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available per processor communication bandwidth for short messages.

- P: the number of processor/memory modules.

As evidenced by experimental data collected on the CM-5 [TM94], this model can accurately predict communication performance when only fixed-sized short messages are sent [CKP+93, CDMS94, LC94]. Many existing parallel machines have special support for long messages which provide a much higher bandwidth than short messages (e.g., IBM SP-2 [BBB+94], Paragon [Pie94], Meiko CS-2 [BCM94], Ncube/2 [SV94]) and this fact is captured by the LogGP model which is a refinement of the LogP model [AISS95]. LogGP extends the LogP model by introducing a new parameter to model long messages:

- G: the *Gap per byte* for long messages, defined as the time per byte for a long message. The reciprocal of G characterizes the available per processor communication bandwidth for long messages.

3.4.2 LogP Complexity

In this section we summarize the LogP complexity of different remap-based implementations of bitonic sort. Let R be the total number of remaps. We consider the total time involved in doing communication per processor. At each remap operation the processors exchange data. For a given processor let V_i be the amount of data transfered at remap R_i. Then the time spent by a processor at remap i in doing communication is:

$$T_{R_i} \quad = \quad L + 2o + (V_i - 1) * \max(g, 2o).$$

In general $2o < g$ and therefore we get:

$$T_{R_i} \quad = \quad L + 2o + g * (V_i - 1).$$

For all the remap operations, the time involved in doing communication (per processor) will be:

$$\begin{aligned} T_{Remap} &= (L+2o)*R+g*(V-R) \\ &= (L+2o-g)*R+g*V. \end{aligned}$$

The above formula shows that the communication time doesn't depend only on the number of communication stages but also on the total number of elements transfered.

In the following we will analyze three remap-based bitonic sort algorithms with respect to the three metrics discussed above:

- number of remaps,
- total volume of elements transfered per processor,
- total number of messages transfered.

The three remapping strategies are:

1. *Blocked Remap* — In this case the first $\lg n$ stages are local and the last $\lg P$ stages require communication (see Section 2.2). We get:

$$\begin{aligned} R &= \frac{\lg P(\lg P+1)}{2}, \\ V &= n*\frac{\lg P(\lg P+1)}{2}. \end{aligned}$$

2. *Cyclic-Blocked Remap* — In this case (see Section 2.3) the first $\lg n$ stages are also local and for the last $\lg P$ stages we periodically remap from blocked to cyclic. We get:

$$\begin{aligned} R &= 2\lg P, \\ V &= 2n\frac{P-1}{P}\lg P. \end{aligned}$$

3. *Smart Remap* — In this case (see Section 3.2) we will use the formulas for the case $\frac{\lg P(\lg P+1)}{2} \leq \lg n$. We get:

$$\begin{aligned} R &= \lg P + 1, \\ V &= n \lg P. \end{aligned}$$

Since we use only short messages the total number of messages sent is the same as the volume of transfered elements, therefore in this case the two metrics are equivalent. As we can see the Smart Remap is optimal for all three metrics and therefore the communication time is minimized for this case. This may not be the case under the LogGP model however, where grouping elements in larger messages can impact the total communication time as we will show in the next section.

3.4.3 LogGP Complexity

The LogGP model captures the fact that elements can be grouped in long messages and sent to their destination processor in only one communication step. We will recompute the formula for the time spent by a processor in doing communication at remap i. The new communication metric that we also take into consideration now is:

- the number of messages transfered

We will denote by M_i the number of messages transfered at remap i by a processor (by M we will denote the total number of transfered messages per processor). Under the same assumption that $2o < g$ we get:

$$T_{R_i} = L + 2o + G * (V_i - M_i) + g * (M_i - 1).$$

Usually $G \ll g$ and therefore by grouping the elements into long messages we obtain a drastic improvement in the total communication time:

$$\begin{aligned} T_{Remap} &= (L + 2o) * R + G * (V - M) + g * (M - R) \\ &= (L + 2o - g) * R + G * V + (g - G) * M. \end{aligned}$$

Under the LogGP model, however, under certain circumstances with respect to the above communication metric the smart remap may in fact perform the worst. Let's compute the total number of messages sent for all three examples like in the previous section.

1. *Blocked Remap* — In this case for each step in the last $\lg P$ stages that require communication (see Section 2.2) processors communicate in pairs. The data on one processor is transfered to the other one and each processor will keep its portion of the data. Thus at each step each processor sends one big message of size n and therefore we get:

$$M = \frac{\lg P(\lg P + 1)}{2}.$$

2. *Cyclic-Blocked Remap* — At each one of the $2\lg P$ remaps each processor sends a message of size n/P to every other processor. Therefore we get:

$$M = 2\lg P(P-1).$$

3. *Smart Remap* — In this case (see Section 3.2) we will use a lower bound on the number of messages. Remember from Lemma 4 that at each remap a processor communicates with other $2^{N_{BitsChanged}} - 1$ processors exchanging messages of size $n/2^{N_{BitsChanged}}$. Considering only the communication at *OutRemaps* (see Section 3.2.1) we get:

$$\begin{aligned} M &\geq \sum_{i=1}^{\lg P}(2^i - 1) + (2^{\lg P} - 1) \\ &\geq 3(P-1) - \lg P. \end{aligned}$$

Observe that with respect to the number of messages sent the *blocked* version is now the best, sending the smallest number of messages. This assumes implicitly that the memory size on each processor is at least twice the data size, otherwise in the blocked case we wouldn't be able to send messages of size n. Although the number of messages sent is very small, the data volume is very big for the blocked version making it unpractical for large data sets and a big number of processors. However, for a small number of processors, for example $P = 2$ we have only one communication step and we send only one message per processor and usually we achieve the best communication time among the three versions.

Given the model parameters L, o, g, G and P we can decide which algorithm is the best (communication-wise) for a given data size n, by plugging in all numbers in the above formulas and comparing the results. However, communication costs also include the time spent during packing/unpacking of the data which can add a substantial overhead to the "real" communication time as we will see in Chapter 5.

Chapter 4

Optimizing Computation

For a wide variety of applications a significant percentage of the total running time is spent during the local computation phases. Tuning the local computation can result in a significant improvement in the total running time of the algorithm. In this chapter we discuss how we optimized the local computation phase on each processor. We start by characterizing the input sequences that we use in optimizing the local computation then we show how we can use only fast sorting routines for this phase, we present an efficient implementation of bitonic merge sort, we show how we can combine the pack and unpack phases with the local computation and finally we discuss the complexity of our methods.

In the case of the bitonic sorting network several optimizations can be performed if we don't resume to simply simulate the behavior of the butterfly network, but instead we rely on the particular format of the input sequences for each computation phase. Since a smart-remap can occur anywhere inside a stage (as opposed to cyclic-blocked where we remap only at the beginning and at step $\lg n$ of a stage) it is necessary to characterize the format of the data array at each column of the bitonic sorting network.

4.1 Local Sorts and Bitonic Merges

In this section we show how we can replace the compare-exchange operations with very fast local sorts. In the next two lemmas we characterize how the data array looks like at each column of the bitonic sorting network (see Definition 3).

Lemma 6 *The data array at the input of stage k, where $1 \leq k \leq \lg N$, consists of $2^{\lg N - k+1}$ alternating increasing and decreasing sorted sequences of length 2^{k-1} (see Figure 4.1).*

Proof: The property results directly from the definition of the bitonic sorting network and the duality of the algorithmic and network view of bitonic sort. Stage k consists of $N/2^k$ alternating bitonic merges of size 2^k ($BM^{\oplus}_{2^k}$ or $BM^{\ominus}_{2^k}$). The output of a bitonic merge is a sorted sequence and the result follows. □

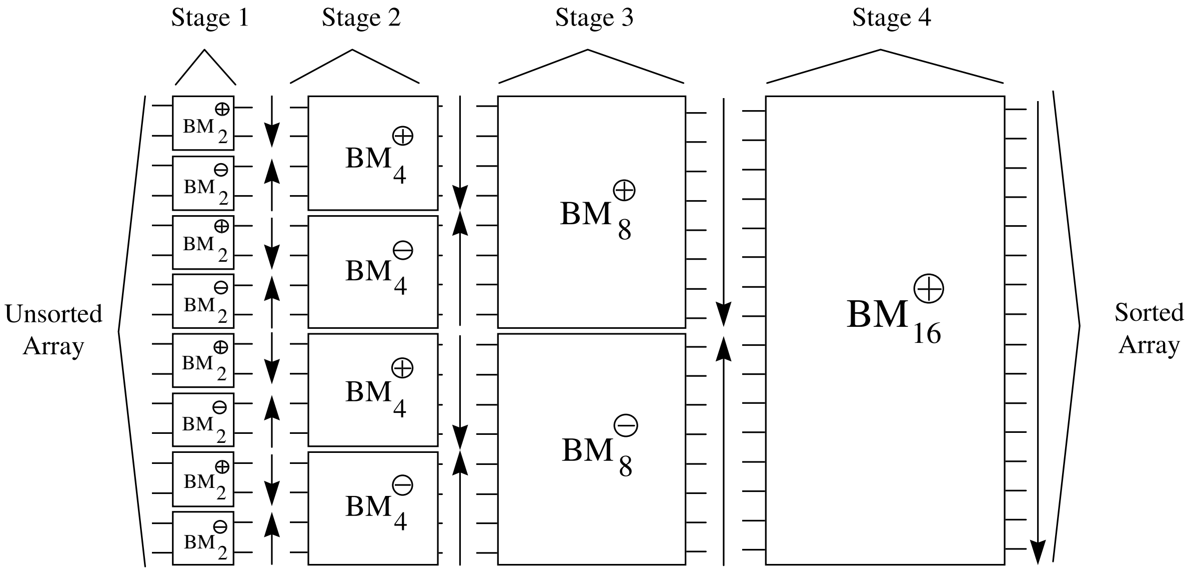


Figure 4.1: *Input of a stage of the bitonic sorting network consists of alternating sorted sequences.*

The previous observation leads to straightforward optimizations (applied in [CDMS94]): the purpose of the first $\lg n$ stages is to form a monotonically increasing or decreasing sequence of n keys on each processor, thus we can replace all these stages with a single, highly optimized local sort. Furthermore, if we use a cyclic-blocked remapping strategy then for the last $\lg P$ stages we can optimize the computation performed for the last $\lg n$ steps of each stage (which execute under a blocked layout) by using a local sort (see Figure 4.2).

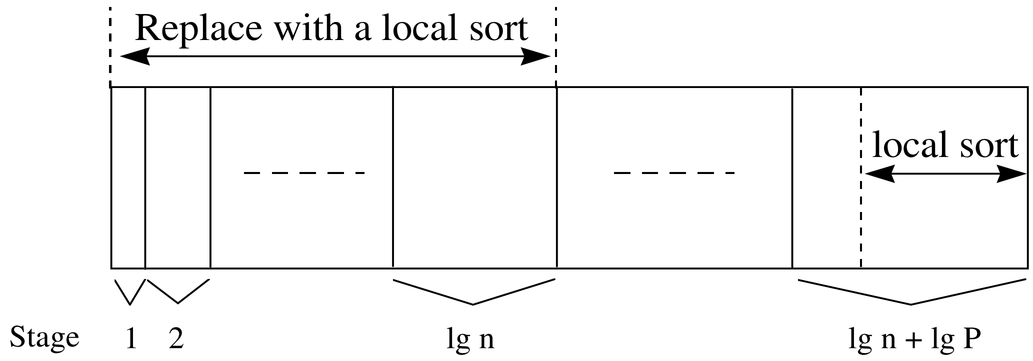


Figure 4.2: *Optimizing local computation by replacing compare-exchange operations with faster local sorts.*

So far we have characterized the format of the data array at the beginning and the end of a stage. The following lemma characterizes the format of the data array at each column inside a stage.

Lemma 7 *The data array at column s of stage k, where $k < s < 0$, consists of $2^{\lg N - s}$ bitonic sequences of length 2^s.*

> **Proof:** The property results directly from the definition of a bitonic split and the fact that each step of the bitonic sorting network performs the equivalent of a bitonic split operation.
>
> At the beginning of stage k (the column number is k) the data array consists of $2^{\lg N - k + 1}$ alternating sorted sequences of length 2^{k-1}. Since two adjacent sorted

sequences form a bitonic sequence we get $2^{\lg N-k}$ bitonic sequences of length 2^k. After s steps, applying a bitonic split at each step, we obtain $2^{\lg N-s}$ bitonic sequences of length 2^s. □

The result of this lemma was applied in [CDMS94] for optimizing the local sort (under the blocked layout) for the last $\lg P$ stages. The observation is that on each processor we have bitonic sequences of length n (from Lemma 7, where column number $= \lg n$), therefore we can replace the local sort with a simpler *bitonic merge sort*.

The following two theorems will show how we can optimize the local computation in the case of a smart layout. Remember that under our remapping strategy we periodically perform $\lg n$ steps locally before remapping again. Depending whether the $\lg n$ steps following the smart-remap cross the boundary between two stages of the bitonic sorting network we derived two types of smart-remaps: *inside* and *crossing*. For each one of those we will have a special way to optimize the computation suggested by the next two theorems.

Theorem 2 (Inside Remap) *For an* Inside Smart-Remap, *the keys assigned to a processor form a bitonic sequence. After performing* $\lg n$ *steps of the bitonic sorting network, this sequence of keys will be sorted.*

Proof: For simplicity we will consider only the keys remapped to processor 0. Consider an inside smart-remap characterized by the 5-tuple (k, s, a, b, t). Remember that for an inside remap we have $a = 0$ and $b = \lg n$, therefore only t characterizes how we select the elements that go on a processor (see Figure 3.7).

The proof has two parts:

- *Proof that the input is a bitonic sequence* - To select the elements that go on processor 0 we pick every 2^t element from the data array at column s of stage $\lg n + k$. From Lemma 7 we know that the first 2^s elements form a bitonic sequence. Since $s = \lg n + t$ every 2^t element we select will be from this sequence. Since a subsequence of a bitonic sequence is still a bitonic sequence, the first part of the theorem follows.
- *Proof that the output is a sorted sequence* - For the output of the inside smart-remap we look at column t of stage $\lg n + k$ in the bitonic sorting network. Similarly, for processor 0, we pick every 2^t element from the

data array at column t, which consists (from Lemma 7) of $2^{\lg N-s}$ bitonic sequences of length 2^s. We are interested only in the first $2^{\lg n+k-t}$ sequences, because for processor 0 we pick the first element of the first $\lg n$ of these bitonic sequences. The important property of these bitonic sequences is that they are all part of the same bitonic merge stage ($BM^{\oplus}_{\lg n+k}$), therefore all the elements of any bitonic sequence are less or equal than the elements of the following bitonic sequence. Selecting an element from each sequence will generate a sorted sequence.

□

Optimizing the computation for this phase is straightforward: all we have to do is to sort a bitonic sequence. We use the very simple *bitonic merge sort*: after the minimum element of the sequence has been found, the keys to its left and right are simply merged. In the next subsection we will show how we can optimize this further by finding the minimum of a bitonic sequence in logarithmic time.

For a crossing smart-remap the pattern of the absolute address is more complicated, however the goal is the same: replace the more expensive simulation of the butterfly with simpler sorts.

Theorem 3 (Crossing Remap) *For the* $\lg n$ *steps following a* Crossing Smart-Remap (k, s, a, b, t) *we distinguish two computation phases (see Figure 3.8):*

- The first a steps after the remap (within stage $\lg n + k$). *Here the input on each processor consists of* 2^b *bitonic sequences of length* 2^a*. At the end of this phase, i.e. at the boundary between stages* $\lg n + k$ *and* $\lg n + k + 1$ *these sequences will be sorted. Additionally, the data on each processor at the end of this phase will consist of two sorted sequences the first one increasing and second one decreasing.*

- The last b steps of the remap (within stage $\lg n + k + 1$). *Here we change the local remap by interchanging the first* b *bits of the local address with the last* a *bits. Under this new layout the input on each processor will consist of* 2^a *bitonic sequences of length* 2^b *and at the end of the phase these will be sorted.*

Proof: For simplicity we will consider again only the case of processor 0. We will consider the two cases separately:

- *The first a steps of the remap (within stage* $\lg n + k$*).* From the pattern of the absolute address for this remap we see that we select every 2^b block of 2^a elements from the data array. From Lemma 7 we know that these are bitonic sequences. At the boundary between stages $\lg n + k$ and $\lg n + k + 1$ the data array consists of sorted sequences of length $2^{\lg n + k}$ (from Lemma 6). Therefore, every 2^b block of size 2^a will represent a sorted sequence. At the end of this phase the last $\lg n + k$ bits of the absolute address are sorted, meaning that locally all the bits except the first one are sorted. Thus at the end of the phase the data array consists of an increasing and then decreasing sequence of elements.
- *The last b steps of the remap (within stage* $\lg n + k + 1$*).* The reason we choose to change the local remap is fairly obvious: From Lemma 6 we know that the data array at the end of stage $\lg n + k + 1$ consists of $2^{\lg P - k - 1}$ alternating increasing and decreasing sorted sequences of length $2^{\lg n + k + 1}$. The result of executing stage $\lg n + k + 1$ is that the last $\lg n + k + 1$ bits of the absolute address are sorted, i.e. nodes with absolute addresses that have the same $\lg n + k + 1$ last bits are in sorted order. We have $b + t = \lg n + k + 1$, therefore the first b steps of stage $\lg n + k + 1$ will sort bits $b + t + 1 \ldots t + 1$ in the absolute address. From the way we remapped the data locally these are the last b bits of the local address. Thus we will have to sort all the 2^a sequences of length 2^b.

  To prove that these sequences are bitonic sequences is also straightforward: previously we have shown that at the beginning of this phase the data array consists of an increasing and then decreasing sequence of elements, i.e. a bitonic sequence (see Figure 4.4). A subsequence of it will still be a bitonic sequence.

□

Using the previous theorem we can optimize the computation by using only bitonic merges to sort the bitonic sequences.

So far we have shown that we can use only local sorts for the local computation phase. However, in the case of a *crossing* remap that is followed by another *crossing* remap we can take the optimizations one step further by using only one local sort to sort the entire data array on the processor.

Figure 4.3 presents two consecutive *crossing* remaps, $Remap_0$ and $Remap_1$ (notations are the same as the ones used in previous proofs and definitions). $Remap_0$ takes place in stage $\lg n + k$ and $Remap_1$ takes place in stage $\lg n + k + 1$. From Lemma 6 we know that the output of stage $\lg n + k + 1$ consists of sorted sequences of length $2^{\lg n + k + 1}$. Since for $Remap_0$, we remap on a new processor every 2^{t-a} block of size 2^a (for a *crossing* followed by a *crossing* we have $t = a + k + 1$), the data array on the processor at the end of the computation phase associated with the remap will consist of 2^b bitonic sequences of size 2^a with the property that all the elements in a bitonic sequence are greater than any of the elements in the bitonic sequences preceding it. The observation is that if we sort all the elements we will obtain 2^b sorted sequences of size 2^a with the property that a sorted sequence of size 2^a has exactly the same elements as its corresponding bitonic sequence (see Figure 4.4).

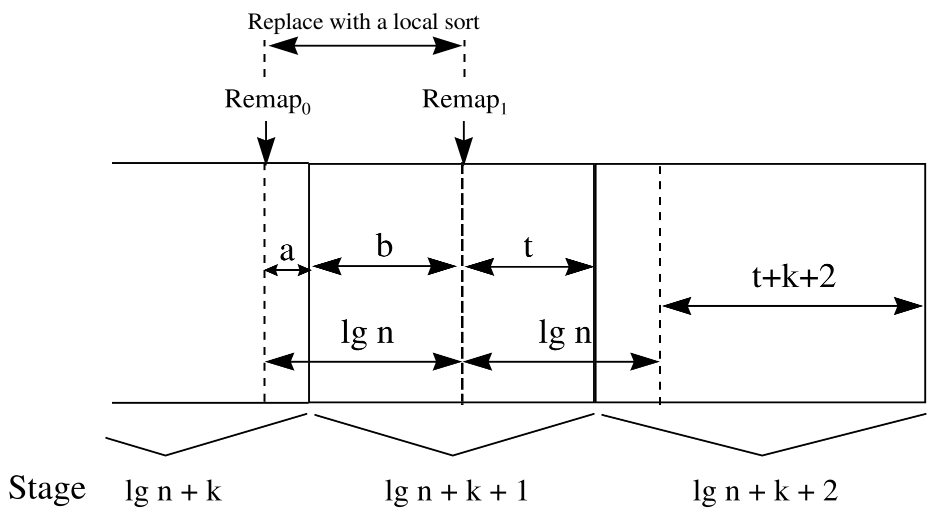


Figure 4.3: *Computation between two crossing remaps can be replaced with a local sort.*

The second important observation here is that for a *crossing* to *crossing* remap every sequence of size 2^a will be remapped on the same processor (because $t > a$) therefore, although we changed their order, elements will remap to the right processor.

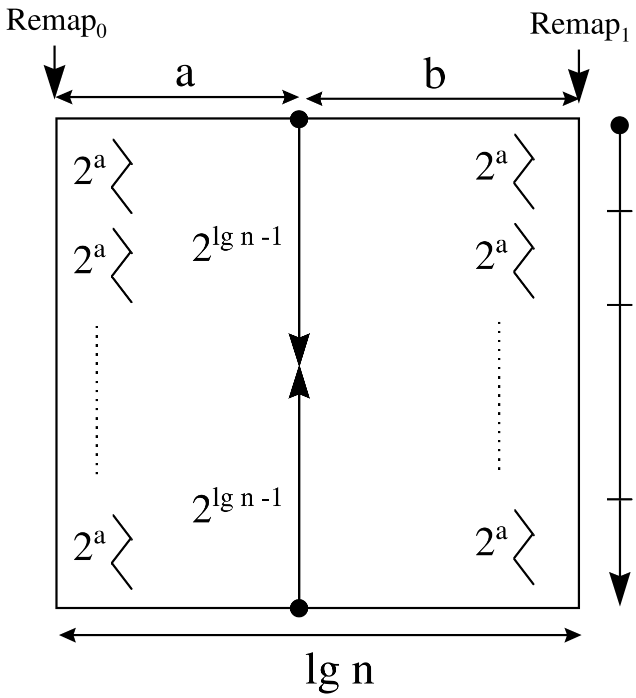


Figure 4.4: *The local computation phase after a crossing remap. The bitonic sequences at the end of the computation have the property that all the elements in a bitonic sequence are greater than any of the elements in the bitonic sequences preceding it.*

Furthermore, as we saw from Theorem 3 the purpose of the first t steps in $Remap_1$ is to generate two sorted sequences. For $Remap_1$ each processor will get every 2^{k+2} block of size 2^t therefore all the blocks coming from the same processor will remap either in the first or the second half of the data array on the destination processor, therefore we can simply sort the two halves (see Figure 4.4). The above property can also be easily verified by looking at the pack and unpack masks for a *crossing* to *crossing* remap from Section 3.3.

Recall from Theorem 2 that for an *inside* remap we also sort the data. Therefore, for all the remaps except a *crossing* remap followed by an *inside* remap we can just sort the data on each processor. This fact turns out to be very important because for usual computations (i.e. large data sets to be sorted) we have

$$\frac{\lg P(\lg P+1)}{2} \quad \leq \quad \lg n,$$

and therefore we have an initial *inside* remap and then only *crossing* remaps (see Section 3.2.1). Thus all we have to do for the local computation phase is to sort the data array on every processor (see Figure 4.5).

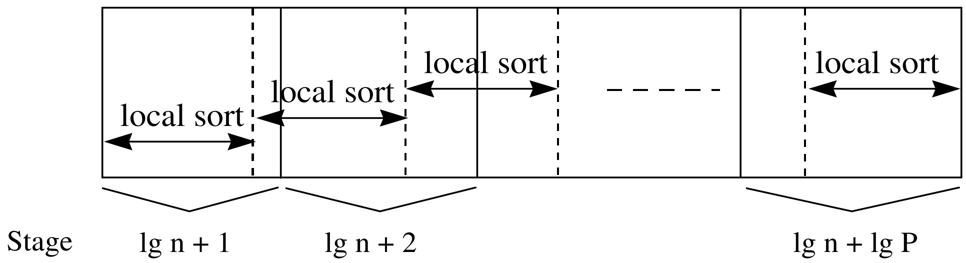


Figure 4.5: *For* $\frac{\lg P(\lg P+1)}{2} \leq \lg n$ *we can use local sorts for each local computation phase.*

The straightforward approach would be to use radix-sort, however, we can take advantage of the special data format (see for example Theorem 2 or Theorem 3) and use faster sorts. Furthermore, in Section 4.3 we will show how we can combine the local sort and the packing and unpacking phases thus eliminating unnecessary data movements and implicitly the overhead associated with these phases.

4.2 Implementing Bitonic Merge Sort

As we saw previously we can replace all compare-exchange operations with local sorts. For random data arrays the best choice would be radix-sort. However, since most of the data sets to be sorted are bitonic sequences we can take advantage of their special properties and implement a sorting algorithm faster than radix-sort. In this section we present an algorithm for sorting bitonic sequences called *bitonic merge sort*.

For a bitonic input sequence, the fastest way to sort it is to use a merge sort instead of simulating the last stage of a bitonic sorting network. This consists of two phases: first the minimum element of the bitonic sequence is found, and second we use mergesort to merge the keys to the left and right of the minimum.

Finding the minimum of a sequence of elements is usually done in linear time. However, in some of the practical situations (like in the case of a bitonic sequence) we can lower this complexity (i.e., find the minimum in logarithmic time). As defined in Definition 1 a bitonic sequence can be viewed (after a circular shift) as a sequence which first increases and then decreases. Therefore we can abstractly represent the sequence under a circular format (see Figure 4.6) which has a maximum and a minimum element. The following algorithm finds the minimum of a bitonic sequence in

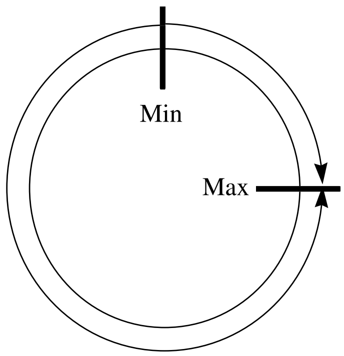


Figure 4.6: *A bitonic sequence can be represented in a circular format.*

logarithmic time.

Algorithm 2 (Find the Minimum of a Bitonic Sequence)

Step 1 - *The algorithm starts by selecting three splitters which break the circular sequence into three parts. Let's denote with a, b, c the three splitters and assume that a is the minimum of the three (see Figure 4.7). Then the minimum of whole sequence cannot be in between b and c, otherwise a will not be the minimum of the three. Therefore we restrict our search to the segments $[b, a]$ and $[a, c]$, where a is the minimum of a, b, c.*

Step 2 - *We select two new splitters x and y to split the intervals $[b, a]$ and $[a, c]$ respectively (see Figure 4.7). Depending on the minimum of (x, a, y) we have 3 possibilities:*

- $min = x$ - *We restrict our search to $[b, x]$ and $[x, a]$*
- $min = a$ - *We restrict our search to $[x, a]$ and $[a, y]$*
- $min = y$ - *We restrict our search to $[a, y]$ and $[y, c]$*

If when we find the minimum of a group of three splitters we have two equal minimums from this point we have to use a sequential search for the minimum on the remaining interval. Otherwise we stop when our search interval consists of only the three splitters and we return the minimum of the three.

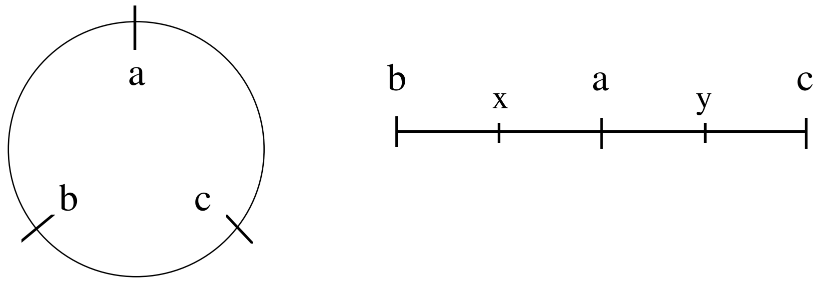


Figure 4.7: *Choosing the splitters when finding the minimum of a bitonic sequence.*

The following lemma characterizes the complexity of our algorithm:

Lemma 8 *Algorithm 2 finds the minimum element of a bitonic sequence without duplicate elements in $O(\log n)$ time.*

Proof: The algorithm recursively applies *Step 2* until the minimum is found. The condition that we don't have duplicate elements guarantees that the minimum of any three splitters is unique so that we can restrict the search to a smaller interval.

Except for the first step the algorithm reduces the searching interval by a factor of two after each splitter selection and comparison. Therefore we have an algorithm with a logarithmic time complexity to find the minimum of a bitonic sequence. □

The above algorithm works in logarithmic time even if we have duplicates, as long as we don't have two equal minimum splitters. We can start finding the minimum using the logarithmic version and we switch to the linear search if we have two equal splitters.

4.3 Combining Computation with Data Packing and Unpacking

In this section we show how we can eliminate the overhead associated with data packing and unpacking by combining these phases with the local computation.

In Section 4.1 we have shown that after an *inside* remap and a *crossing* remap followed by a *crossing* remap we simply sort the data on each processor. After every local sort we pack the data for each processor into a long message and send it so its destination (see Section 3.3). Thus, for the new remap stage following one of the above mentioned stages, the data array on each processor will consist of a series of sorted sequences. Recall from Section 3.2.1 that at each remap operation processors exchange data in groups of 2^k if the $\lg n$ steps following the remap will end within stage $\lg n + k$. Another important property is that the first half of these processors generated increasing sorted sequences and the second half decreasing sorted sequences (this results from Lemma 6), thus we will have 2^{k-1} increasing sequences and 2^{k-1} decreasing sequences. Therefore we can eliminate the unpacking phase by taking advantage of the input's special format and sort the data using a fast p-way merge sort.

The overhead associated with packing can be eliminated similarly by incorporating this phase within the last step of the sorting procedure. Since we are using only radix-sort, simple merges and bitonic merges this can be done easily by computing a *pack index* for every element that has been sorted and assigning the element to the its location in the packed message instead of its position in the sorted sequence. For this all we need is the *pack mask* defined in Section 3.3.

For our implementation we have modified the original versions of radix-sort, the two way-merge and the bitonic merge to perform the sort and packing in a single step. The above optimizations are illustrated in Figure 4.8 for a crossing remap.

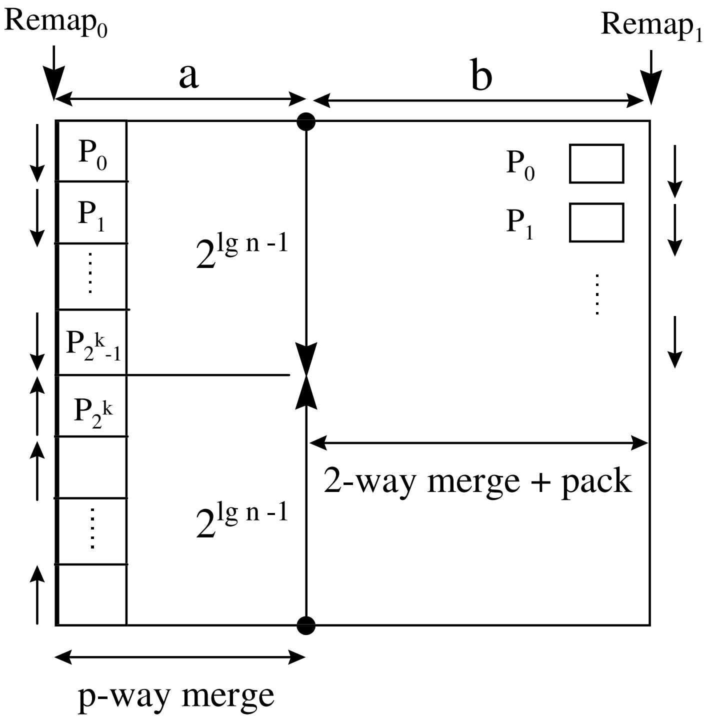


Figure 4.8: *Combining computation with data packing and unpacking. The labels on the left represent the processor number where the packed data comes from, the labels on the right represent the processor number where the new packets will be sent.*

4.4 Computation Complexity

In this section we focus on the complexity of the local computation. Without any optimization the complexity of simulating $\lg n$ steps of the bitonic sorting networks would be $O(n \log n)$. The lemma following characterizes the complexity of the bitonic merge sort and is a direct consequence of the definition of the bitonic sequence (Definition 1):

Lemma 9 *Sorting a bitonic sequence takes $O(n)$ time, where n is the length of the bitonic sequence, versus $O(n \log n)$ for the naive algorithm.*

For the last $\lg P$ stages by using only p-way and bitonic merges for the local computation phases we have reduced the computation complexity to $O(n)$. For the first $\lg n$ stages since the keys are in a specified range we used radix-sort which also takes $O(n)$ time.
To summarize for the entire parallel bitonic sort we distinguish the following phases:

- *local radix sort* for the first $\lg n$ stages of the bitonic sorting network followed by
- succession of *smart remaps* and *local merges.*

Since radix sort, the p-way merge and the bitonic merge are all linear on the input size and we have $O(\lg P)$ computation phases, the complexity of the local computation for the entire bitonic sort algorithm is $O(n \lg P)$.

Chapter 5

Experimental Results

In this chapter we describe our experimental platform and environment, we present experimental data collected for different implementations of the parallel bitonic sort algorithm, we analyze the impact of long messages on the algorithm efficiency and finally we compare our algorithm against other parallel sorts.

5.1 The Meiko CS-2 Parallel Computer

Our experimental platform is the Meiko CS-2 which consists of Sparc based nodes connected via a fat tree communication network [HM93]. Running a slightly enhanced version of the Solaris 2.3 operating system on every node, it closely resembles a cluster of workstations connected by a fast network. Our machine is a 64 node CS-2. Each node contains a 40 MHz SuperSparc processor with 1 MB external cache and 32 MB of main memory. The SuperSparc is three-way super-scalar processor, and thus achieves a peak rate of 120 MIPS. Each node in a CS-2 contains a special communications co-processor, the Elan processor, which connects the node to the fat tree. CS-2's communications co-processor has a DMA engine which allows efficient implementations of bulk transfers.

For our implementation we used an optimized version of the Split-C parallel language implemented on Meiko CS-2 on top of Active Messages [vECGS92, SS95].

5.2 The Split-C Programming Language

We have implemented our algorithm in Split-C [CDG+93], a parallel extension of the C programming language designed for large, distributed-memory multiprocessors. The goal of the Split-C language was to combine the most valuable aspects of shared memory with the most valuable aspects of message passing and data parallel programming within a coherent framework. The language follows an SPMD (single program multiple data) model, provides a shared global address space, global pointers and spread arrays and offers assignment operations that allow the programmer to overlap communication with computation and avoid unnecessary responses. Especially useful for our implementation were the bulk transfer operations which allowed us to send large messages of variable size.

Split-C is currently implemented on a wide variety of parallel machines including CM-5, IBM SP-2, Meiko CS-2, Cray T3D, cluster of workstations, making it especially attractive for architecture independent applications.

5.3 Performance Measurements

In this section we present and compare the measurements for our implementation and two other implementations of parallel bitonic sort previously used in important studies of parallel sorting algorithms.

The first one is the Split-C implementation of bitonic sort used in one of the first studies of efficient parallel sorting algorithms [BLM+91]. The algorithm uses a local radix-sort for the first $\lg n$ stages then for each subsequent stage $\lg n + k$, where $1 \leq k \leq \lg P$, executes k communication steps and at each one of them exchanges data between pairs of processors and simulates a merge step of the bitonic sorting network, then uses local radix-sort again for the remaining $\lg n$ steps of the stage. We will call this algorithm the *Blocked-Merge* bitonic sort.

The second implementation is the *Cyclic-Blocked* version presented in Section 2.3. The computation performed under the cyclic layout consists of bitonic merges, and under the blocked layout of local radix sorts. This implementation was used to study parallel bitonic sort in another major study of parallel sorting algorithms [CDMS94] and was by our knowledge the fastest implementation of parallel bitonic sort. We will call this algorithm the *Cyclic-Blocked* bitonic sort.

Our implementation uses the smart remapping strategy presented in Section 3.2. The local computation is also optimized as presented in Chapter 4. We will call our algorithm the *Smart* bitonic sort.

In all of the above algorithms the communication phase is implemented using long messages. We use random, uniformly-distributed 32-bit keys.[1] We measured and compared the total execution time and the execution time per key on 2 to 32 processors and for 16K to 1M keys per processor.

Table 5.1 presents the execution time per key for the above implementations and Table 5.2 presents the total running time for the three versions on 32 processors. Figure 5.1 shows the total execution times for the three algorithms for 32 processors and Figure 5.2 plots the data in Table 5.1.

Figure 5.3 shows the total sorting time and the speedup curve for sorting 1M keys on 2 to 32 processors. Figure 5.4 presents the breakdown of the communication and computation phases of our bitonic sort algorithm. As we can see, when we increase the number of elements, a higher percentage of the total execution time is spent during the local computation phases. This is due to cache misses suggesting that further improvement can be achieved by tuning the local computation for better cache behavior.

[1] Actually, our random number generator produces numbers in the range 0 through $2^{31} - 1$.

Keys/proc (in K)	Blocked-Merge	Cyclic-Blocked	Smart
128	1.07	0.68	0.52
256	1.19	0.75	0.51
512	1.26	0.89	0.53
1024	1.25	0.86	0.59

Table 5.1: *Execution time per key (in μs) for different implementations of the bitonic sort algorithm on 32 processors.*

Keys/proc (in K)	Blocked-Merge	Cyclic-Blocked	Smart
128	5.52	2.85	2.18
256	10.04	6.35	4.26
512	21.14	14.96	8.95
1024	42.03	28.58	20.01

Table 5.2: *Total execution time (in seconds) for different implementations of the bitonic sort algorithm on 32 processors.*

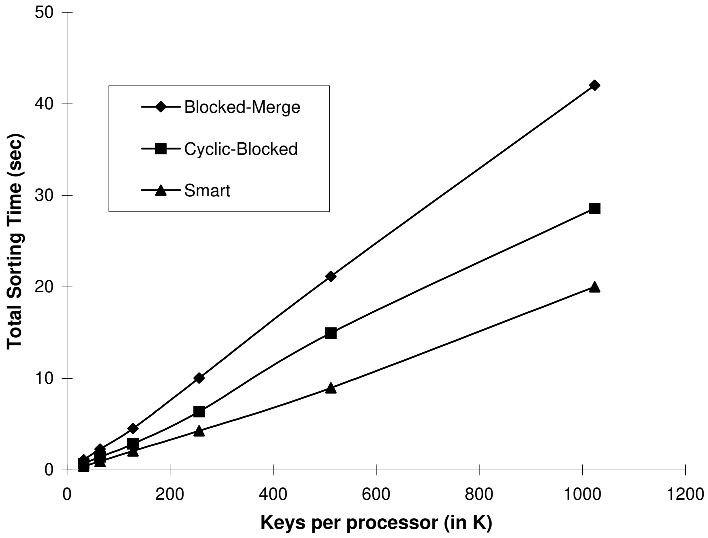


Figure 5.1: *Total execution time (in seconds) for different implementations of the bitonic sort algorithm on 32 processors.*

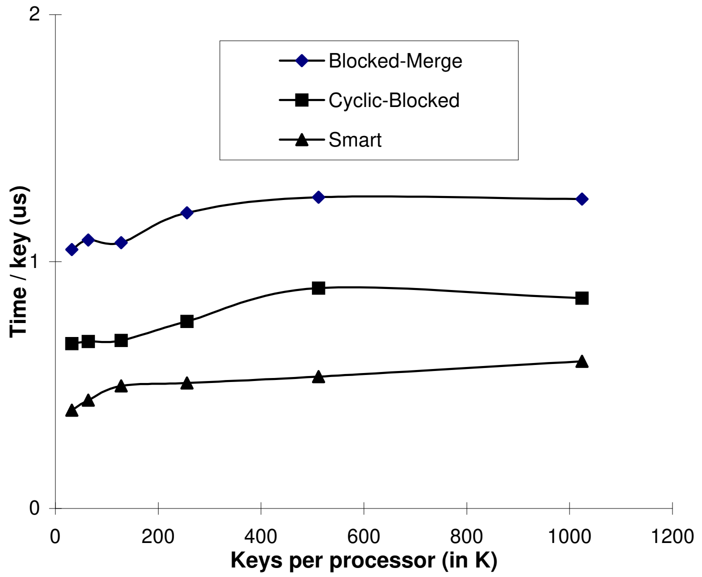


Figure 5.2: *Execution time per key (in μs) for different implementations of the bitonic sort algorithm on 32 processors.*

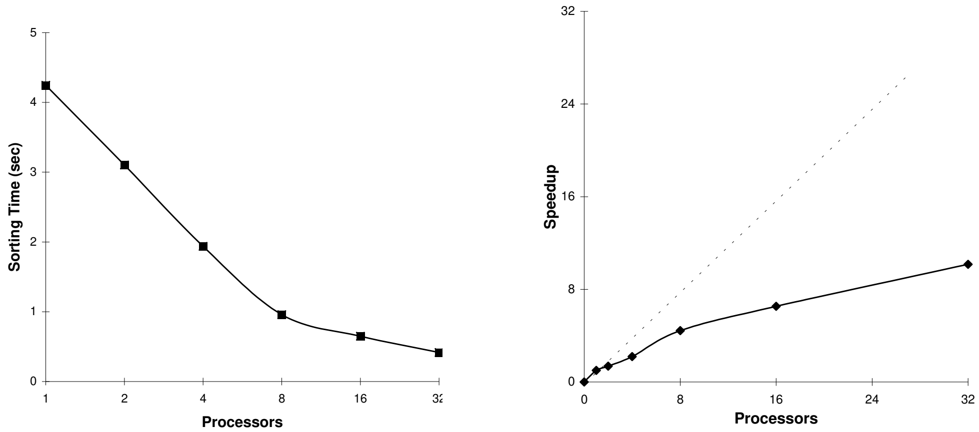


Figure 5.3: *Total sorting time and speedup curve for sorting 1 million keys on Meiko CS-2.*

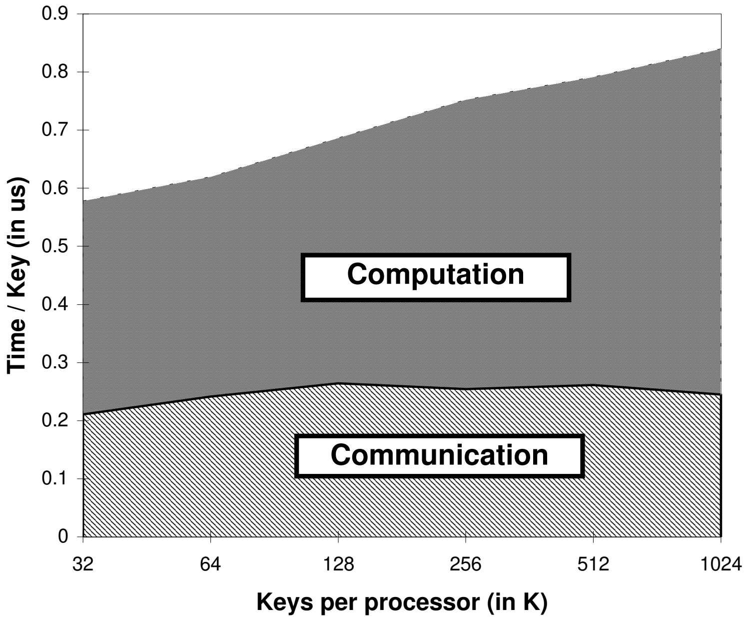


Figure 5.4: *Breakdown of the communication and computation phases of the bitonic sort algorithm on 16 processors.*

5.4 Effect of Long Messages

The experimental data presented above was collected only for algorithms in which long messages were used. As we showed in Section 3.3 intermediary phases of packing and unpacking the data into/from long messages are required, however, we can eliminate the overhead associated with these by incorporating them into the local computation as we have shown in Section 4.3. Even in the case when we don't combine local computation with the pack and unpack phases, although we add more overhead, the total time involved in doing communication is substantially better than in the case when we use short messages. In the following we will contrast two versions of our bitonic sort implementation: one that uses long messages and does not combine the pack and unpack phases with local computation and one that uses short messages. The local computation part is identical for both versions.

Table 5.3 shows the communication time per key for the two versions (short and long messages) of the smart remap based algorithm. Table 5.4 presents the breakdown of the communication phase for the long messages version. As we can see the pack and unpack phases represent approximately 80 % of the total communication time.

Figure 5.5 presents the communication time per key for the two versions and Figure 5.6 plots the breakdown from Table 5.4.

Keys/proc (in K)	Short Messages	Long Messages
128	13.23	0.98
256	13.25	1.09
512	13.26	1.12
1024	13.74	1.21

Table 5.3: *Execution time per key (in μs) for the short messages and the long messages versions of the bitonic sort algorithm on 16 processors.*

Keys/proc (in K)	Packing	Transfer	Unpacking
128	0.35	0.15	0.15
256	0.37	0.15	0.15
512	0.38	0.16	0.14
1024	0.38	0.16	0.13

Table 5.4: *Breakdown of the execution time per key (in μs) for the communication phase for the long messages version on 16 processors.*

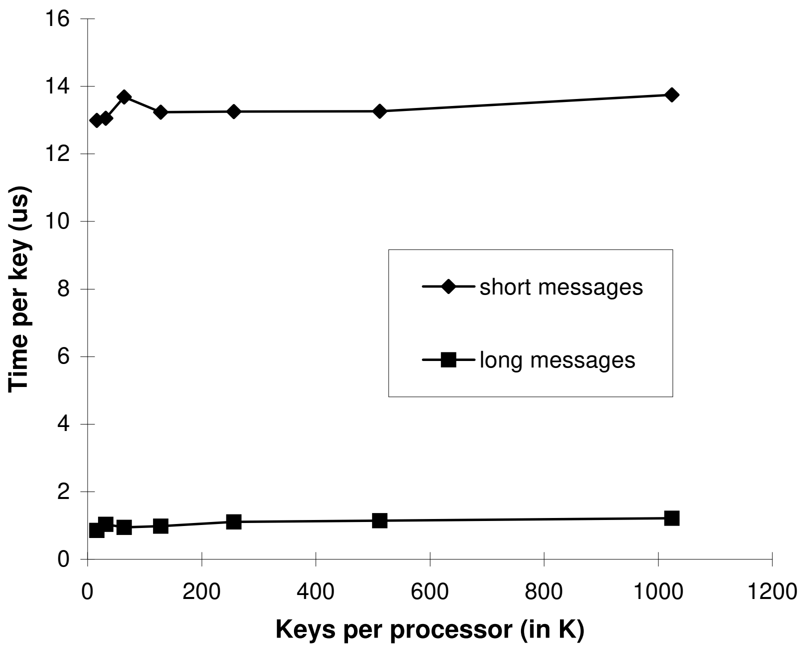


Figure 5.5: *Execution time per key (in μs) for the short messages and the long messages versions of the bitonic sort algorithm on 16 processors.*

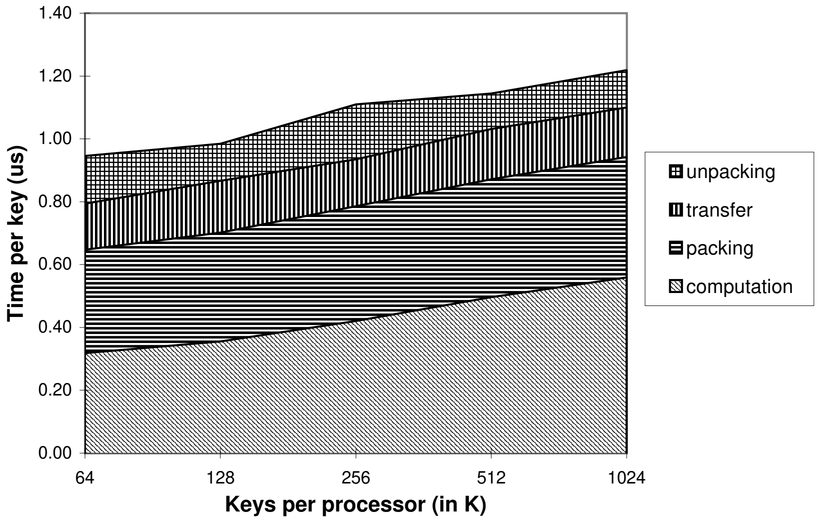


Figure 5.6: *Breakdown of the execution time per key (in μs) for the long messages version on 16 processors.*

5.5 Comparison with other Parallel Sorts

We have previously mentioned two studies [BLM+91, CDMS94] that compared the efficiency of several general-purpose parallel sorting algorithms on modern parallel machines (e.g. CM-2, CM-5). In both studies three parallel algorithms were analyzed and compared: bitonic sort, radix sort and sample sort. The comparison criteria were different, however with respect to the total execution time sample sort was the fastest followed by radix and then bitonic sort. Bitonic sort performed well only for small data sets and for a small number of processors.

We have compared our implementation of bitonic sort with two highly optimized versions of sample and radix sort. The algorithms for sample and radix sort used in [BLM+91, CDMS94] were using short messages and were slower than our long message implementation of bitonic sort. The sample and radix sort versions that we have used were implemented using long messages [AISS95] and they were by our knowledge the fastest Split-C implementations of sample and radix sort.

Figure 5.7 shows the execution time per key for sample, radix and bitonic sort on 16 processors and Figure 5.8 on 32 processors. As we can see for 16 processors our implementation of bitonic sort performs better than radix sort. On 32 processors bitonic sort is still better than radix sort for up to 256K elements per processor. Sample sort is still the clear winner, but for a small number of processors and for small data sets bitonic sort may in fact perform better. Furthermore the performance of sample sort is strongly dependent on the initial distribution of the keys: a low entropy input set may lead to unbalanced communication and contention. Bitonic sort on the other hand is oblivious to the input distribution. These comparisons suggests that for a small number of processors and for small data sets bitonic sort might be the fastest choice.

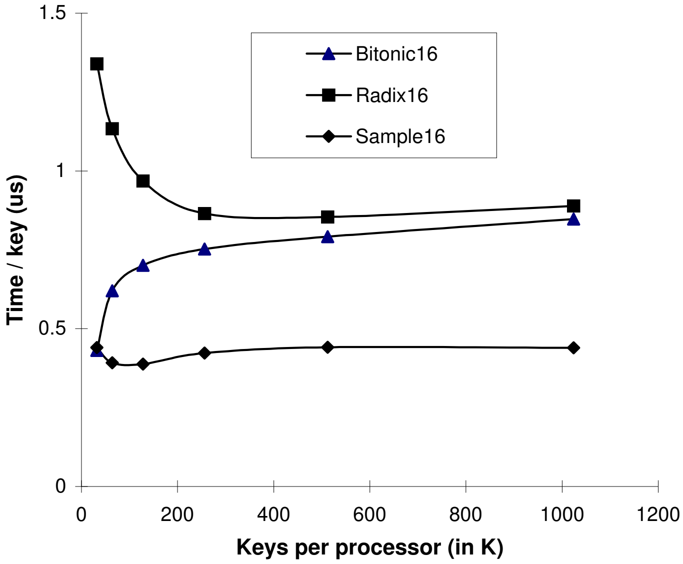


Figure 5.7: *Execution time per key (in μs) for sample, radix and bitonic sort on 16 processors.*

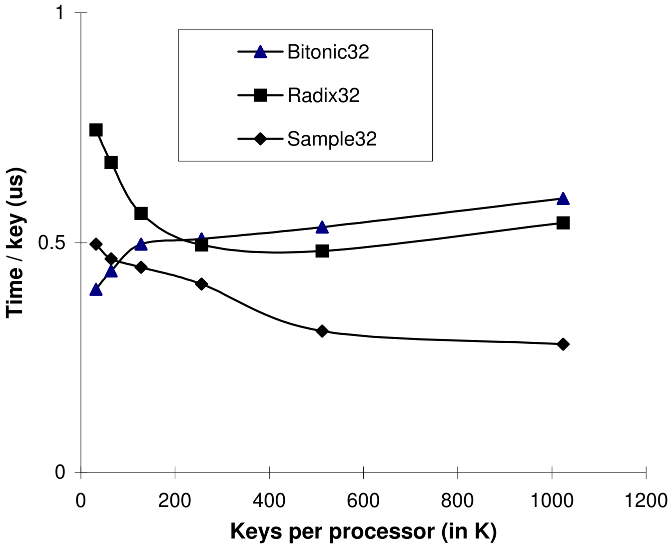


Figure 5.8: *Execution time per key (in μs) for sample, radix and bitonic sort on 32 processors.*

Chapter 6

Related Work

Bitonic sort and sorting networks have received special attention since Batcher [Bat68] in 1968 showed that fine-grained parallel sorting networks can sort in $O(\lg^2 n)$ time using $O(n)$ processors. Since then a lot of effort has been directed at fine-grain parallel sorting algorithms (e.g. see [BDHM84, Já92, KR90, Rei93]). In 1983 it was shown [AKE83] that n elements can be sorted in $O(\lg n)$ time with an $O(n \lg n)$-sized network. In 1985 Leighton [Lei85] extended this result to show that one can produce an $O(n)$-node bounded-degree network capable of sorting in $O(\lg n)$ steps, based upon an algorithm called *column sort*. In 1988 Cole [Col88] gave simple methods for optimal sorting in the CREW and EREW PRAM models in $O(\lg n)$ time using $O(n)$ processors based on a *cascade mergesort* using arrays. Therefore in a fine-grain parallel model like the one mentioned above, one can sort optimally.

These algorithms are not optimal, however when implemented under more realistic models of parallel computation. The later make the "realistic" assumption that the data size N is larger than the number of processors P. Now the goal becomes to design a general-purpose parallel sort algorithm that is the *fastest* in practice. One of the first important studies of the performance of parallel sorting algorithms was conducted by Blelloch, Leiserson et al. [BLM+91] which compared bitonic, radix and sample sort on CM-2. Several issues were emphasized like space, stability, portability and simplicity.

These comparisons were augmented by a new study by Culler et al. ([CDMS94, Dus94]). Column sort was included and a more general class of machines was addressed by formalizing the algorithms under the LogP model. All algorithms were implemented in Split-C (a parallel language which provides a straightforward machine independent programming system) [CDG+93] making them available to be ported and analyzed across a wide variety of parallel architectures. The conclusion of this study was that an "optimized" data layout across processors was a crucial factor in achieving fast algorithms. Optimizing the local computation was another major factor that contributed to the overall performance of the algorithms. The study also showed that by a careful analysis of the algorithm under a realistic parallel computation model we can eliminate design deficiencies and come up with efficient implementations .

A more recent study [AISS95] showed that a large class of parallel algorithms (and in particular sorting algorithms) can be substantially improved when re-designed under a new model of parallel computation (LogGP) which captures the fact that modern parallel machines have support for long message transfers, therefore achieving a much higher bandwidth than short messages. By careful re-design of the algorithms improvements of more than an order of magnitude were achieved over previous implementations of radix sort, FFT, sample sort. One technique used (which is also described in this thesis) was packing the data into long messages at the sending site and unpacking it at the receiving site.

It is also worth to mention the relationship of bitonic sort to column sort. Like bitonic sort, column sort [Lei85] alternates between local sort and key distribution phases, but only four phases of each are required. Two of the communication phases are similar to cyclic-to-blocked and blocked-to-cyclic remaps discussed in Chapter 2, the others are just a one-to-one communication. Like the cyclic-blocked bitonic sort, column sort requires that $N \geq P^3$.

Chapter 7

Conclusion and Future Work

In this thesis we have analyzed optimizations that can be applied to the parallel bitonic sort algorithm. We have designed and implemented a remap-based bitonic sort algorithm that uses the smallest possible number of data remaps. Besides minimizing the communication requirements, local computation has also been optimized by taking advantage of the special format of the data sets. For this we have shown that local computation can be entirely replaced with faster local sorts. We have also presented a general method for remapping the data which is useful in applications that need to take full advantage of the high network bandwidth by grouping data into long messages.

Furthermore, we have analyzed bitonic sort under two realistic models of parallel computation, LogP and LogGP. We have considered three fundamental metrics that influence the communication time of a parallel algorithm:

- the number of data communication steps,
- the total volume of data transfered per processor,
- the number of messages sent across the network.

We have shown that the total communication time is dependent upon all the above metrics and minimizing just one of them is not sufficient to obtain a communication optimal algorithm.

Our experimental results have shown that our implementation is much faster than any previous implementation of parallel bitonic sort. We have compared our algorithm with other parallel sorts (radix sort, sampler sort) and for a small number of processors or small data sets our algorithm is the fastest. We have also shown the impact of long messages on the overall performance of the algorithm and we have presented methods for performing a remap between two data layouts using long messages. Furthermore, we have shown how the overhead associated with the data packing and unpacking phases can be eliminated by combining them with the local computation phase.

Although we have explained how one can optimize communication by obtaining the smallest possible number of data remaps several avenues for future research remain:

- derive the layout that optimizes the total volume of elements transfered during the execution of the algorithm,
- derive the layout that optimizes the number of messages sent across the network,
- overlap computation and communication.

Altogether, our optimizations show a definite improvement over previous approaches, however, further refinements can be applied to the sorting algorithm that will improve its performance such as:

- combine the unpacking, local sorts and the packing phase into a single local computation step (our implementation combines these three phases but in two steps),
- expensive data movements performed during the local computation phase can be reduced by carefully analyzing the data format and changing the input pattern for the sorting routines,

- optimize for better cache and register behavior (this can be done by following a similar strategy to the data remaps).

Overall, as we have shown and as other studies have shown, bitonic sort is one of the best choices when the number of processors is small and for small data sets. Among other advantages the algorithm is very space efficient, contention free and well balanced for different key distributions.

Although, most probably, our optimizations will not generate the "fastest" parallel sorting algorithm they are applicable in a large variety of applications (not only parallel). We can mention here the FFT which is based on a butterfly network (i.e. a stage of the bitonic sorting network) or the omega networks (for which similar remapping techniques can be applied). The same techniques can be applied to improve the memory locality (i.e. cache and register access) for different large array computations like recursive FFT or blocked matrix multiply.

Finally, another observation and a possible research topic is that our technique of remapping the data, given a data pattern configuration, in such a way that data accesses are minimized is applicable in any hierarchical memory model. Since accesses across different layers of the hierarchy are very expensive, given the "communication pattern" (i.e. memory access pattern) we can derive data remaps such that we maximize the ratio of local accesses to remote accesses.

Overall, we hope that the techniques presented in this thesis will be further refined and applied for a larger class of algorithms. We feel that the applicability area of our methods is larger than parallel computing and it extends to memory hierarchy models and numerical computations involving data sets under various layouts.

Bibliography

[ABK95] M. Adler, J. W. Byers, and R. M. Karp. Parallel sorting with limited bandwidth. In *Proceedings the Symposium on Parallel Algorithms and Architectures*, July 1995.

[ACS90] A. Aggarwal, A. K. Chandra, and M. Snir. Communication Complexity of PRAMs. In *Theoretical Computer Science*, March 1990.

[AISS95] A. Alexandrov, M. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP model — One step closer towards a realistic model for parallel computation. In *7th Annual Symposium on Parallel Algorithms and Architectures*, July 1995.

[AKE83] M. Ajtai, Komlós K., and Szemerédi E. Sorting in $c \log n$ parallel steps. In *Combinatorica*, March 1983.

[Bat68] K. Batcher. Sorting Networks and their Applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, volume 32, 1968.

[BBB+94] V. Bala, J. Bruck, R. Bryant, R. Cypher, P. de Jong, P. Elustondo, D. Frye, A. Ho, C-T. Ho, G. Irwin, S. Kipnis, R. Lawrence, and M. Snir. The IBM external user interface for scalable parallel systems. *Parallel Computing*, 20(4), April 1994.

[BCM94] E. Bartson, J. Cownie, and M. McLaren. Message passing on the Meiko CS-2. *Parallel Computing*, 20(4), April 1994.

[BDHM84] D. Bitton, D. J. DeWitt, D. K. Hsiao, and J. Menon. A Taxonomy of Parallel Sorting. Technical Report TR84-601, Cornell University, Computer Science Department, April 1984.

[BLM+91] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the Symposium on Parallel Architectures and Algorithms*, 1991.

[BN86] G. Bilardi and A. Nicolau. Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared Memory Machines. Technical Report TR86-769, Cornell University, Computer Science Department, August 1986.

[CDG+93] D. E. Culler, A. Dusseau, S. C. Golstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of Supercomputing*, November 1993.

[CDMS94] D. E. Culler, A. Dusseau, R. Martin, and K. E. Schauser. Fast Parallel Sorting under LogP: from Theory to Practice. In T. Hey and J. Ferrante, editors, *Portability and Performance for Parallel Processing*. Wiley, 1994.

[CKP+93] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[Col88] R. Cole. Parallel merge sort. *SIAM J. Comput*, 17(4), 1988.

[Dus94] A. Dusseau. Modeling Parallel Sorts with LogP on the CM-5. Technical Report UCB/CSD 94/829, UC Berkeley, May 1994.

[HM93] M. Homewood and M. McLaren. Meiko CS-2 Interconnect Elan-Elite Design. In *Proc. of Hot Interconnects*, August 1993.

[HS82] Z. Hong and R. Sedgewick. Notes on merging networks. In *Proceedings of the 14th Annual Symposium on Theory of Computing*, May 1982.

[Já92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, Mass., 1992.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin Cummings, 1994.

[Knu73] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison Wesley, Reading Massachusetts, 1973.

[KR90] R. M. Karp and V. Ramachandran. Parallel Algorithms for Shared-Memory Machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* Elsevier Science Publishers, 1990.

[KRS90] C. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. In *Theoretical Computer Science*, 1990.

[LC94] L. T. Liu and D. E. Culler. Measurements of Active Messages Performance on the CM-5. Technical Report UCB/CSD 94-807, CS Div., UC Berkeley, May 1994.

[Lei85] F. T. Leighton. Tight bounds on complexity of parallel sorting. *IEEE Transactions on Computers*, 34(4), 1985.

[Lei92] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufman, 1992.

[Pie94] P. Pierce. The NX message passing interface. *Parallel Computing*, 20(4), April 1994.

[PY88] C. H. Papadimitriou and M. Yannakakis. Towards an Architecture-Independent Analysis of Parallel Algorithms. In *Proceedings of the Twentieth Annual ACM Symposium of the Theory of Computing.* ACM, May 1988.

[Qui94] M. J. Quinn. *Parallel Computing. Theory and Practice.* Mc-Graw Hill, 1994.

[Rei93] J. H. Reif. *Synthesis of Parallel Algorithms.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[SS95] K. E. Schauser and C. J. Scheiman. Experience with Active Messages on the Meiko CS-2. In *9th International Parallel Processing Symposium*, April 1995.

[Sto71] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Computer*, C-20(2), February 1971.

[SV94] M. Schmidt-Voigt. Efficient parallel communication with the nCUBE 2S processor. *Parallel Computing*, 20(4), April 1994.

[TM94] L. W. Tucker and A. Mainwaring. CMMD: Active messages on the CM-5. *Parallel Computing*, 20(4), April 1994.

[Val90a] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8), August 1990.

[Val90b] L. G. Valiant. Parallel Algorithms for Shared-Memory Machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* Elsevier Science Publishers, 1990.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, May 1992.