


Optimizing parallel bitonic sort

Mihai F Ionescu

Proceedings 11th International Parallel Processing Symposium

Related papers

[Download a PDF Pack](#) of the best related papers 



[A generalization of Amdahl's law and relative conditions of parallelism](#)

Gianluca Argentini

[The superblock: An effective technique for VLIW and superscalar compilation](#)

Scott Mahlke

[Membrane systems with surface objects](#)

Gabriel Ciobanu, Bogdan Aman

Optimizing Parallel Bitonic Sort

Mihai Florin Ionescu and Klaus E. Schauser

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{mionescu, schauser}@cs.ucsb.edu

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.

This paper studies bitonic sort on modern parallel machines which are relatively coarse grained and consist of only a modest number of nodes, thus requiring the mapping of many data elements to each processor. Under such a setting optimizing the bitonic sort algorithm becomes a question of mapping the data elements to processing nodes (data layout) such that communication is minimized. We developed a bitonic sort algorithm which minimizes the number of communication steps and optimizes the local computation. The resulting algorithm is faster than previous implementations, as experimental results collected on a 64 node Meiko CS-2 show.

1 Introduction

Sorting is a popular Computer Science topic which receives much attention. As a parallel application, the problem is especially interesting because it fundamentally requires communication as well as computation [ABK95] and is challenging because of the amount of communication it requires. Parallel sorting is 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, such as PRAM or network-based models, where the ratio of memory to processors is relatively small [BDHM84, J92, KR90, Lei92, Rei93, Qui94]. Later research has shown, however, that processor-to-processor communication is the most important bottleneck in parallel computing [ACS90, CKP⁺93, KRS90, PY88, Val90a, Val90b, AISS95]. Thus efficient parallel algorithms are more likely to be achieved on coarse-grain parallel systems and in most situations algorithms originally developed for PRAM-based models are substantially redesigned.

One of the earliest parallel sorting algorithms is Bitonic Sort [Bat68], which is represented by a sorting network consisting of multiple butterfly stages of increasing size. The bitonic sorting network was the first network capable of sorting n elements in $O(\lg^2 n)$ time and 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, BN89], and recent implementations and evaluations 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 [CDMS94, BLM⁺91].

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. Therefore the network size grows proportionally to the input size. 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. Under such a setting optimizing a parallel algorithm becomes a question of optimizing communication as well as computation.

We derive a new data layout which allows us to perform the smallest possible number of data remaps. The basic idea is to locally execute as many steps of the bitonic sorting network as possible. We show that for the last $\lg P$ stages of the bitonic sorting network — which usually require communication — the maximum number of steps that can be executed locally is $\lg \frac{N}{P}$ (N is the data size, P is the number of processors). Our algorithm remaps the data such that it always executes $\lg \frac{N}{P}$ before remapping again, thus executing the smallest possible number of remap operations.

Compared with previous approaches our algorithm executes less communication steps and also 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] and collect experimental results on a 64 node Meiko CS-2. 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. Finally, we compare our implementation of bitonic sort against other parallel sorts.

2 Bitonic Sort

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

Definition 1 (Bitonic Sequence) *A bitonic sequence is a sequence of values a_0, \dots, 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.*

On a bitonic sequence we can apply the operation called *bitonic split* which halves the sequence in two bitonic sequences such that all the elements of one sequence are smaller than all the elements of the other sequence (for details see [KGGK94]). Thus, given a bitonic sequence we can recursively obtain shorter bitonic sequences using bitonic splits, until we obtain sequences of size one, at which point the input sequence is sorted. 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 (known as *bitonic merging network*, see Figure 1).

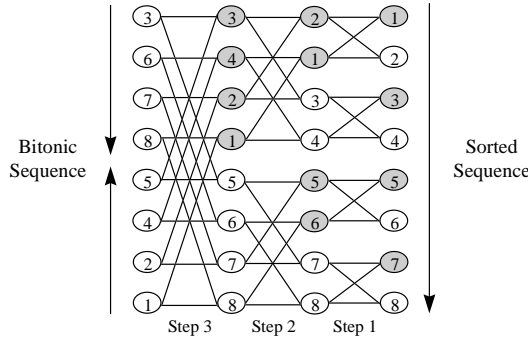


Figure 1: *Butterfly structure of an increasing bitonic merge of size $N = 8$ (we denote this with BM_8^\oplus). A shaded node designates an address where the minimum of the two keys is placed, the unshaded node designates an address where the maximum of the two keys is placed.*

Definition 2 (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).*

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 1), 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 $s, \dots, 0$. 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 3).

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 $(s, c+1, \bar{r}_c)$, where $\bar{r}_c = r \oplus 2^c$ (i.e. r and \bar{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. Basically, the bitonic sorting network implements the bitonic merges described previously.

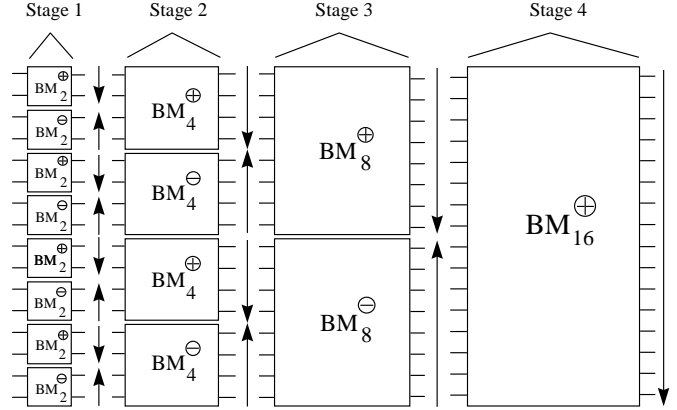


Figure 2: *Block structure of a bitonic sorting network of size $N = 16$. With BM_k^\oplus and BM_k^\ominus we denote increasing, respectively decreasing, bitonic merging networks of size k . The arrows indicate the monotonic ordered sequence, with the arrowhead pointing towards the largest key.*

2.1 Naive and Improved Data Layouts

A straightforward parallel implementation is a naive one: simply simulate the compare-exchange steps in the butterfly network using a *blocked data 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 $n = N/P$. Under a blocked layout, the first $\lg n$ stages execute completely local. For subsequent stages $\lg n + k$, the first k steps require communication while the last $\lg n$ steps are completely local. Another possible data layout is the *cyclic data layout*. A cyclic layout for mapping N keys on P processors assigns the i -th key to the $(i \bmod n)$ -th processor, where $n = N/P$. 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 of the stage 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.

One efficient data placement which minimizes the communication requirements is to switch between different data layouts so that all compare-exchange operations execute locally. Therefore, we can reduce the communication requirements by periodically remapping the data from a blocked layout to a cyclic layout and vice versa. 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 stages that require communication (the last $\lg P$ stages). This approach was suggested in [CKP⁺93, CDMS94] and used for efficient implementations of parallel algorithms based on the butterfly network such as FFT or bitonic sort.

3 Optimizing Communication

As we saw from the cyclic-blocked implementation a good data distribution can dramatically reduce the communication require-

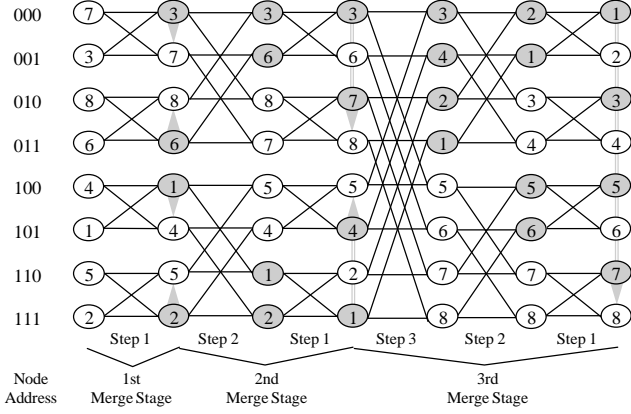


Figure 3: A bitonic sorting network of size $N = 8$. A row of nodes represents an address containing one of the keys. Each node compares two keys, as indicated by the edges and selects either the maximum or the minimum.

ments. The main result presented in this section is that we derive an algorithm which executes the smallest possible number of data remaps and therefore we minimize the number of communication steps.

We need to introduce some new notation: the *absolute address* of a node ($\lg N$ bits long) represents the row number of the node in the bitonic sorting network; at a remap we move nodes across processors and therefore each node also has a *relative address* which consists of two parts: the first $\lg P$ bits represent the processor number and the last $\lg n$ bits the local address of the node after the remap. The following lemma is based on the absolute and relative address representation (for proof and details see [Ion96]):

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).*

3.1 Deriving the Optimal Data Layout

The previous lemma shows that there is a fundamental limitation on how much the communication part of the bitonic sort algorithm can be optimized. A remap strategy that executes exactly $\lg n$ steps locally before remapping again generates the smallest possible number of remaps. We can thus reformulate the problem as: Given the tuple $(stage, step)$, which uniquely identifies a column of the bitonic sorting network, how to remap the elements at this point in such a way that the next $\lg n$ steps of the bitonic sorting network are executed locally? The essential observation is that the execution of $\lg n$ steps of the bitonic sorting network requires comparing only elements that differ in exactly $\lg n$ bits of the absolute address.

We consider two cases depending whether we cross a stage during the $\lg n$ steps following the remap operation. The first case is illustrated in Figure 4 where we have an *inside remap* ($s \geq \lg n$). Here all the steps following the remap are inside the stage where the remap occurs and the bits that are compared in the absolute address are $s \dots s - \lg n + 1$. In the case when we cross a stage (see Figure 5) 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 compare bits $s \dots 1$ and $\lg n + k + 1 \dots k + s + 2$. In both cases we get an absolute and relative address bit pattern. Nodes whose addresses match the shaded parts of the pattern are remapped on the same processor and the next $\lg n$ steps of the bitonic sorting network execute locally. The following definition and lemma formalize our findings:

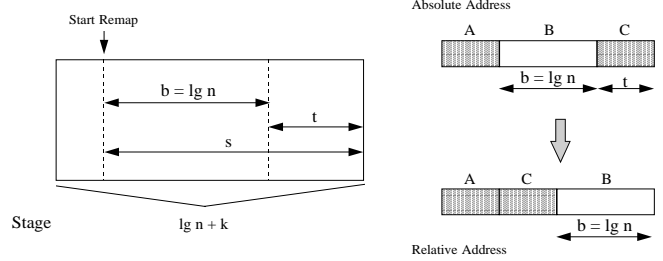


Figure 4: *Inside Remap* and the corresponding absolute and relative address bit pattern. The absolute address specifies the row number of the bitonic sorting network, the relative address specifies the processor (the shaded part) and local offset within the processor after the remap.

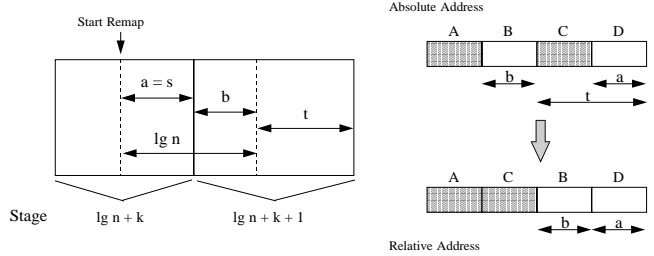


Figure 5: *Crossing Remap* and the corresponding absolute and relative address bit pattern.

Definition 3 (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:

$$a = \lg n, b = 0, t = \lg n.$$

Given the absolute address of a node, the relative address where the node is remapped is computed as presented in Figure 4 (for $s \geq \lg n$) and Figure 5 (for $s < \lg n$).

Data Layout	Number of remaps	Number of elements transferred	Number of messages
Blocked	$\frac{\lg P(\lg P+1)}{2}$	$n \frac{\lg P(\lg P+1)}{2}$	$\frac{\lg P(\lg P+1)}{2}$
Cyclic-Blocked	$2 \lg P$	$2n \frac{P-1}{P} \lg P$	$2 \lg P(P-1)$
Smart	$\lg P + 1$	$n \lg P$	$3(P-1) - \lg P$

Table 1: Communication complexity for three different data layouts.

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

The previous lemmas allow for a simple formulation of our parallel bitonic sorting algorithm:

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 $\lg P$ 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.*

3.2 Communication Complexity Analysis

Our smart data layout minimizes the number of communication steps, but the total communication time of the 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: the total number of remaps, the total number of elements transferred (volume), and the total number of messages transferred.

We study three versions of bitonic sort algorithm using three different data layouts: the *blocked*, *cyclic-blocked* and the *smart* data layouts. We analyze the communication complexity of the algorithm with respect to the three metrics (the total volume and number of messages are considered per processor). The formulas for all three metrics are summarized in Table 1. For the *smart* data layout we considered the practical case when $\frac{\lg P(\lg P+1)}{2} \leq \lg n$ and in the case of number of messages we use a lower bound. Observe that with respect to the total number of elements transferred and the number of remaps the *smart* data layout version is the best. We refer the interested reader to [Ion96] for a more thorough analysis on how these three abstract metrics determine the actual communication complexity under the LogP and LogGP models of parallel computation.

4 Optimizing Computation

In this section we show how we optimize the local computation by replacing the compare-exchange operations with very fast local sorts.

Lemma 3 *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 2).*

This 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 radix sort.

Lemma 4 *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 .*

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 4, where column number = $\lg n$), therefore we can replace the local radix sort with a simpler and much faster *bitonic merge sort*: after the minimum element of the sequence has been found, the keys to its left and right are simply merged.

The following two theorems show how we can optimize the local computation in the case of a smart layout.

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 is sorted.*

Optimizing the computation in this case is straightforward; all we have to do is to sort a bitonic sequence using the simple *bitonic merge sort*.

Theorem 3 (Crossing Remap) *For the $\lg n$ steps following a Crossing Smart-Remap (k, s, a, b, t) there are two computation phases for which the following holds (see Figure 5):*

- 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 are sorted. Furthermore, the data on each processor at the end of this phase consists of two sorted sequences, the first one increasing and second one decreasing.*
- The last b steps of the remap (within stage $\lg n + k + 1$): *After a local remap which interchanges the first b bits of the local address with the last a bits, the input on each processor consists of 2^a bitonic sequences of length 2^b and at the end of the phase these are sorted.*

In this case we can optimize the computation by using multiple bitonic merges to sort bitonic sequences. However, for 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 each processor. In the following we give a brief justification (for details see [Ion96]): The first observation is that if we sort all the elements we obtain 2^b sorted sequences of size 2^a with the property that every sorted sequence of size 2^a has exactly the same elements as its corresponding bitonic sequence obtained if we would simulate the sorting network. The second important observation is that for a *crossing* to *crossing* remap every sequence of size 2^a is remapped on the same processor, therefore, although we changed their order, elements remap to the right processor. Recall from Theorem 2 that for an *inside* remap we also sort the data, thus, for all the remaps except a *crossing* remap followed by an *inside* remap we can just sort the data on each processor.

4.1 Implementing Bitonic Merge Sort

In the following we show how we can optimize *bitonic merge sort* by presenting an algorithm that finds the minimum element of a bitonic sequence without duplicate elements in logarithmic time. Because a bitonic sequence can be viewed (after a circular shift) as a sequence which first increases and then decreases, we can abstractly represent it under a circular format (see Figure 6) which has a maximum and a minimum element.

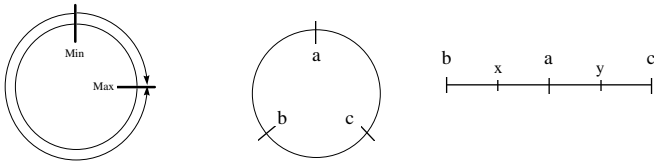


Figure 6: Circular representation of a bitonic sequence (left) and splitter selection (right).

Algorithm 2 (Minimum of a Bitonic Sequence)

Step 1 - The algorithm starts by selecting three splitters which break the circular sequence into three equal parts. Let's denote with a, b, c the three splitters and assume that a is the minimum of the three (see Figure 6). Then the minimum of the whole sequence cannot be in between b and c , otherwise a would not be the minimum of the three. Therefore we restrict our search to the segments $[b, a]$ and $[a, c]$ (a is the minimum of a, b, c) which form a bitonic sequence and we recursively apply Step 2.

Step 2 - We select two new splitters x and y to split the intervals $[b, a]$ and $[a, c]$ respectively (see Figure 6). 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 we find two equal minimums (of a group of three splitters) we use a sequential search to find 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.

The above algorithm works in logarithmic time even if we have duplicates, as long as we don't have two equal minimum splitters.

4.2 Computation Complexity

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

Lemma 5 *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 first $\lg n$ stages since the keys are in a specified range we used radix-sort which takes $O(n)$ time. For the last $\lg P$ stages by using only bitonic merges we have reduced the computation complexity to $O(n)$ for each stage. Since we have $O(\lg P)$ computation phases, the complexity of the local computation for the entire bitonic sort algorithm is $O(n \lg P)$.

5 Experimental Results

Our experimental platform is the Meiko CS-2 which consists of Sparc based nodes connected via a fat tree communication network. For our implementation we used an optimized version of the Split-C parallel language [CDG⁺93] implemented on Meiko CS-2 on top of Active Messages [vECGS92, SS95].

We present and compare the measurements for our implementation (called *Smart* bitonic sort) and two other implementations of parallel bitonic sort previously used in important studies of parallel sorting algorithms. The first is the parallel bitonic sort implementation from [BLM⁺91] (which was reimplemented in Split-C). 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 call this algorithm the *Blocked-Merge* bitonic sort. The second implementation is the *Cyclic-Blocked* version [CDMS94] presented in Section 2.1. The computation performed under the cyclic layout consists of bitonic merges, and under the blocked layout of local radix sorts. All algorithms are implemented in Split-C and the communication phase uses long messages. We use random, uniformly-distributed 32-bit keys (actually, our random number generator produces numbers in the range 0 through $2^{31} - 1$). 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. Figure 7 shows the total execution times and the execution time per key for the three algorithms for 32 processors.

We also compared our implementation of bitonic sort with two highly optimized versions of sample and radix sort (implemented using long messages [AISS95]). Figure 8 shows the execution time per key per processor for sample, radix and bitonic sort on 16 and 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 performs 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 is the fastest choice. The performance comparison of parallel sorting algorithms, radix sort

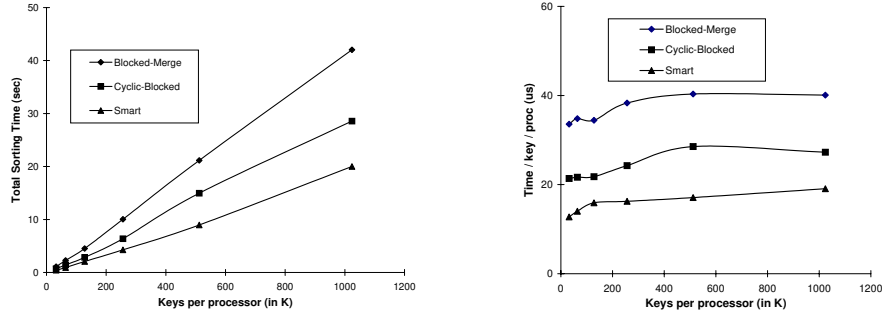


Figure 7: Total execution time (left) and execution time per key (right) for different implementations of the bitonic sort algorithm on 32 processors.

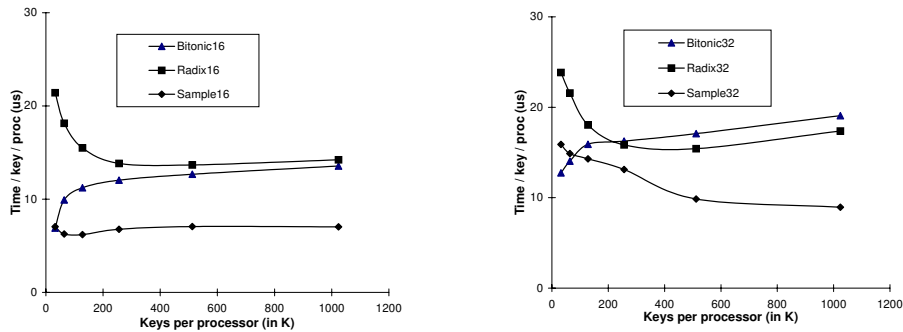


Figure 8: Execution time per key per processor for sample, radix and bitonic sort on 16 processors (left) and 32 processors (right).

and sample sort in particular, is also the subject of more recent studies [BHJ96, HJB96]. Notably, the author’s implementation of sample sort is invariant over the set of input distributions.

6 Related Work

Bitonic sort and sorting networks have received special attention since Batcher [Bat68] 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, Jaj92, KR90, Rei93, AKS83, Lei85, Col88]).

Many of these fine-grained 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 much 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]. 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 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 the LogGP model 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 improvements of more than an order of magnitude were achieved over previous implementations of radix sort and sample sort. These observations are also emphasized in [BHJ96, HJB96] where the authors’ main focus is on efficient implementation of parallel sorting algorithms under a realistic computation model with a strong accent on portability. Another study of various parallel sorting algorithms, including bitonic sort, on SIMD machines is [BW97] which compares implementations of different deterministic oblivious methods for large values of N/P .

7 Conclusion and Future Work

In this paper we have analyzed optimizations that can be applied to the parallel bitonic sort algorithm. We have designed and implemented a remap-based 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. Furthermore, we have analyzed three fundamental metrics that influence the communication time of a parallel algorithm (the number of remaps, the total number of transferred elements, and the number of messages) and we have shown that the total communication time is dependent upon all three 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 and for a small number of processors or small data sets our algorithm is faster than other parallel sorts such as radix or sample sort.

Overall, we hope that our techniques 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.

Acknowledgments

This work was supported by the National Science Foundation under NSF CAREER Award CCR-9502661. Computational resources were provided by NSF Instrumentation Grant CDA-9216202. We would like to thank David Bader, Chris Scheiman and the anonymous reviewers for their valuable feedback on this paper.

References

- [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. Schauer, 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.
- [AKS83] M. Ajtai, K. Komlós, and E. Szemerédi. 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.
- [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.
- [BHI96] D.A. Bader, D.R. Helman, and J. Jájá. Practical Parallel Algorithms for Personalized Communication and Integer Sorting. *ACM Journal of Experimental Algorithmics*, 1(3), 1996.
- [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.
- [BN89] G. Bilardi and A. Nicolau. Adaptive Bitonic Sorting: An Optimal Parallel Algorithm for Shared Memory Machines. *SIAM Journal on Computing*, April 1989.
- [BW97] K. Brockmann and R. Wanka. Efficient Oblivious Parallel Sorting on the MasPar MP-1. In *Proceedings of the 30th Hawaii International Conference on System Sciences*, 1997.
- [CDG+93] D. E. Culler, A. Dusseau, S. C. Goldstein, 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. Schauer. 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. Schauer, 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.
- [HJB96] D. R. Helman, J. Jájá, and D. A. Bader. A New Deterministic Parallel Sorting Algorithm With an Experimental Evaluation. Technical Report CS-TR-3670 and UMIACS-TR-96-54, University of Maryland, August 1996.
- [HS82] Z. Hong and R. Sedgewick. Notes on merging networks. In *Proceedings of the 14th Annual Symposium on Theory of Computing*, May 1982.
- [Ion96] M. F. Ionescu. Optimizing Parallel Bitonic Sort. Master's thesis, also available as Technical Report TRCS96-14, Department of Computer Science, University of California, Santa Barbara, July 1996.
- [Jáj92] 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.
- [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.
- [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*. McGraw Hill, 1994.
- [Rei93] J. H. Reif. *Synthesis of Parallel Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1993.
- [SS95] K. E. Schauer 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.
- [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. Schauer. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, May 1992.