Efficient Oblivious Parallel Sorting on the MasPar MP-1[*]

Klaus Brockmann[†] Rolf Wanka[‡]

TR-96-042

Abstract

We address the problem of sorting a large number N of keys on a MasPar MP-1 parallel SIMD machine of moderate size P where the processing elements (PEs) are interconnected as a toroidal mesh and have 16KB local storage each. We present a comparative study of implementations of the following deterministic oblivious sorting methods: Bitonic Sort, Odd-Even Merge Sort, and FastSort. We successfully use the *guarded* split&merge operation introduced by Rüb. The experiments and investigations in a simple, parameterized, analytical model show that, with this operation, from a certain ratio N/P upwards both Odd-Even Merge Sort and FastSort become faster on average than the up to the present fastest, sophisticated implementation of Bitonic Sort by Prins. Though it is not as efficient as Odd-Even Merge Sort, FastSort is to our knowledge the first method specially tailored to the mesh architecture that can be, when implemented, competitive on average with a mesh-adaptation of Bitonic Sort for large N/P.

[*]Supported by DFG-Sonderforschungsbereich 376 "Massive Parallelität". Accepted for presentation on the 30th Hawaii International Conference on System Sciences (HICSS), 1997.

[†]Heinz Nixdorf Institute, Paderborn University, 33095 Paderborn, Germany, email: brockm@hni.uni-paderborn.de.

[‡]International Computer Science Institute 1947 Center Street, Berkeley, CA 94704-1198, USA, email: wanka@icsi.berkeley.edu. Supported by DFG Leibniz Grant Me 872/6-1 and by EU ESPRIT Long Term Research Project 20244 (ALCOM-IT). On leave from Dept. of Mathematics and Computer Science, Paderborn University, 33095 Paderborn, Germany, email: wanka@uni-paderborn.de.

1 Introduction

The problem. Sorting is one of the most investigated problems in computer science. In the area of parallel computing, sorting is also a classical topic. Its roots can be traced back to the Fifties [Knu73, p. 244]. Richards' bibliography [Ric86] covers the extensive literature until 1986. In many parallel algorithms, parallel sorting is one of the subroutines that determine the overall performance. E. g., it is used in applications like the computation of convex hulls, parallel data bases, and certain image-processing methods, to name a few. Many parallel sorting circuits and other sorting methods on networks are described in Leighton's book [Lei92].

One type of parallel computer is the SIMD (Single Instruction, Multiple Data) machine, in contrast to the MIMD (Multiple Instruction, Multiple Data) type of machines. In SIMD machines, all processing elements (PEs) execute in one time step on their local data the same operation that is sent to them by an external control unit or are idle. Typically, SIMD machines have a large number of PEs, but the amount of local memory is modest. The time to set up a communication between PEs is usually small compared to the set-up times on MIMD machines. Well-known examples of SIMD machines are the MasPar MP-1 and the Thinking Machines CM-2. In this paper, we report on implementations of three deterministic oblivious sorting algorithms on a small MasPar MP-1 with 2048 PEs each with 16KB memory.

Let N be the number of keys that have to be sorted, and let P be the number of PEs. The method that, for $N \gg P$, usually leads to the fastest runtime on MIMD machines is Samplesort (see next paragraph). However, it has per-processor storage requirements that scale with the number of PEs. The requirement that the complete set of so-called splitters be available at every PE can be problematic for parallel machines in which P is large, but the amount of local memory is modest. In this case, there is a need for efficient in-place sorting algorithms, i. e., algorithms that require additional storage per PE of size $O(1)$ only. Appropriate algorithms working reasonably well in this case are those that are based on sorting circuits (comparator networks [Knu73]). Here, it is known ahead of time and independent of the sequence of inputs which pairs of PEs perform a conditional exchange during each time step. Such sorting algorithms are called *oblivious* . Because every PE stores more than one key, the comparators can be replaced by split&merge operations [Knu73, p. 241] (see Section 3). Note that we still call an algorithm oblivious when it replaces the conditional exchanges with split&merge operations. The split&merge operations are fixed, but their execution time may vary.

Previous work. Blelloch *et al.* [BLM$^+$91] present a comparative study of a variety of implementations of parallel sorting algorithms on a Thinking Machines CM-2 consisting of 65536 one-bit PEs. Comparative studies for MIMD machines can be found in [DCSM96, DGL$^+$94, WW96]. For N/P large, Samplesort [FM70] results in the fastest implementations.

Prins [Pri90] has implemented Batcher's Bitonic Sort [Bat68] on a MasPar MP-1. This is the up to the present fastest known implementation of a deterministic oblivious sorting algorithm on such a machine. The other famous sorting circuit due to Batcher, Odd-Even Merge Sort [Bat68], has not been implemented so far. Hightower *et al.* [HPR92] have introduced a randomized method they call B-Flashsort that is related to Samplesort. Here it can happen that some PEs eventually hold more than N/P keys so that the restriction of a small local memory leads to load-balancing phases during the sorting. In a recent paper, Zheng *et al.* [ZCZ96] report on an implementation of an in-place sorting scheme dubbed ZZ-sort.

Rüb proves [Rüb95] for Odd-Even Merge Sort the phenomenon that, on average, nothing has to be exchanged in most of the split&merge operations, which leads to guarded split&merge operations.

An overview of parallel mesh-sorting algorithms is given by Chlebus and Kukawka [CK90]. Unfortunately, implementations of these algorithms on meshes are slower than adaptations of Bitonic Sort.

Contribution of this paper. In this paper, we describe implementations of the deterministic parallel sorting algorithms Bitonic Sort, Odd-Even Merge Sort, and a multidimensional method called FastSort on a MasPar MP-1 of moderate size, i. e., consisting of 2048 PEs each with 16KB local memory. The PEs are interconnected as a 32×64 torus. These algorithms are oblivious. Note that they do not rely on the representation of the keys (like, e. g., Radixsort). Guarded split&merge operations are used. Because Prins' source code is available, we can compare our implementations to the implementations done by Prins [Pri90] on the same machine. It turns out that, for certain values of N/P, Odd-Even Merge Sort and FastSort become faster on average than Prins' implementation of Bitonic Sort. Our implementation shows that Odd-Even Merge Sort with guarded split&merge operations results in very good runtimes on a SIMD machine, as well as it was shown on a MIMD machine [WW96]. Fastsort is an sorting algorithm for d-dimensional meshes. There is a trade-off between the number of split&merge operations (depending on d) and the time that is spent by routing keys. To our knowledge, FastSort is the first sorting algorithm directly designed for meshes that can for large N/P outperform adaptations of Bitonic Sort on average when implemented.

Organization. In Section 2, we describe in short the MasPar MP-1 architecture, and we give a simple analytical model. The necessary basic routines are introduced in Section 3. We deal with Bitonic Sort, Odd-Even Merge Sort and FastSort in Section 4.

2 The MasPar MP-1

The architecture. All algorithms discussed in this paper have been implemented on a massively parallel SIMD machine of the MasPar series MP-1, which consists of a front end and a data parallel unit (DPU). E.g., see Fig. 1. The front end is a standard UNIX workstation (DECstation 5000) and renders the user access to the DPU. The DPU is the massively parallel subsystem of the MasPar and consists of two major parts: the processing element (PE) array, performing the parallel calculations, and the array control unit (ACU), which controls the PE array and provides a global memory which is shared by all PEs. The ACU broadcasts the instructions and shared variables via the ACU-PE-Bus to the PEs and also performs operations on the scalar data in the parallel program which is stored in the global memory. In our configuration, the PE array has $P = 2048$ PEs, each having a local

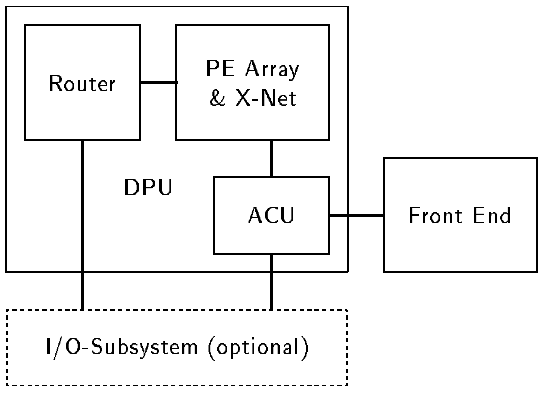


Figure 1: Architecture of the MasPar MP-1.

memory with a capacity of 16 KB. The PEs are arranged as a 32×64 two-dimensional torus, with each PE connected to its eight nearest neighbors (toroidal wrap). At any given moment, communication using this interconnection network (called the X-Net) is only possible in one of the eight given directions, where each PE may send data across the link of the corresponding direction and receive data across the one opposite. In addition, the distances between the communicating processors must all be equal; if it is greater than one, the data is transferred in a *store-and-forward* fashion. Another communication mechanism between the PEs is provided by the Global Router, a multistage crossbar network, which allows the user to carry out arbitrary communication patterns. One has to note, however, that the set-up time for Global Router communication is significantly greater than that of the X-Net and that only one input and one output link exists for each non-overlapping 4×4 matrix or *cluster* of the PE array. Thus, communication operations are executed sequentially whenever more than one incoming connection and one outgoing connection per cluster is required.

A simple model. Since the MasPar is a SIMD machine with a global clock and synchronous execution of each instruction, it seems to be appropriate to develop an exact, but simple model for the purpose of runtime predictions. In the context of the oblivious sorting algorithms discussed in this paper, one has to determine the execution times of the fundamental operations which are comparisons, memory movements, and communication operations. For performance measurements and runtime predictions, we have chosen 32-bit integer keys without associated data records as input to the sorting algorithms (see Section 3). Thus, comparisons and memory movements are done by register operations and load/store instructions. Additionally, the algorithms make use of some instructions for loop control and address calculations. Execution times of all these instructions are given in detail in the original MasPar documentation [Mas93b]. Thus, and for brevity's sake, we will restrict ourselves to giving only the equations for the communication times of our implementations. These are for the X-Net (t_X) and the Router (t_R), respectively:

$$\begin{aligned} t_X(\mathrm{sf}_X, \mathit{dist}) &= \mathrm{sf}_X \cdot (2.9 \cdot \mathit{dist} + 5.4) \quad [\mu\mathrm{s}] \\ t_R(\mathrm{sf}_R) &= \mathrm{sf}_R \cdot 13.8 + 3.0 \quad [\mu\mathrm{s}] \end{aligned}$$

The arguments for these equations are as follows: sf_X is the sequencing factor of X-Net communication. Recall that at any moment X-Net communication is possible in one direction only. Thus, in general (but not always) this factor will be two, e. g., when PEs have to exchange keys. If the distance between the communicating PEs (*dist*) is greater than one, the keys are transferred in a store-and-forward fashion. Thus, communication time becomes linear with this parameter. sf_R is the sequencing factor of Router communication, which is the maximum number of sending (receiving) PEs per cluster. In the given context of oblivious sorting algorithms, $\mathrm{sf}_R \leq 16$ for each communication.

3 Basic Operations

In this section, we describe some basic routines which are commonly used by our parallel sorting algorithms. In the following, we assume that the number of keys exceeds the number of processors, i. e., $N > P$, and that the keys are distributed evenly among all processors, i. e., each processor holds $n := N/P$ keys. For performance analysis and experiments, we choose 32-bit integer keys with no associated data records. The keys are generated by a pseudo random number generator, provided as part of the MasPar Programming Language standard libraries. Prins also used this generator for testing his implementations.

Internal sorting: Odd-Even Merge Sort. Given the fact that each processor holds more than just one key, we replace the compare/exchange operations of the parallel sorting algorithms by split&merge operations (see below). For an efficient implementation of this operation, it is required that the keys are stored locally as sorted sequences. We can meet this requirement by initially sorting the locally stored keys with a fast sequential algorithm. We choose a sequential implementation of the Odd-Even Merge Sorting network for the solution of this task. The main advantage of this method is that it is an oblivious algorithm, i. e., all local memory accesses of the PEs can be controlled by the ACU which is significantly faster than performing the address calculations by the PEs locally. This is the reason why Odd-Even Merge Sort is faster (for the considered number of keys) than other methods which are theoretically optimal, e. g., heapsort or mergesort [Knu73]. Because we have a SIMD machine, an implementation of quicksort is not reasonable, because all PEs have to wait for completing the slowest recursive call before they can resume. The number of comparators of Odd-Even Merge Sort is $\frac{1}{4} n \log n (\log n - 1) + n - 1$. One step consists of loading two operands into registers, a comparison of these, and then storing them back into the memory. Since the total time for these operations amounts to 26.3μs, the time consumption of the sequential implementation is:

$$t_{SeqSort}(n) = \left(\tfrac{1}{4} n \log n (\log n - 1) + n - 1\right) \cdot 26.3 \quad [\mu\mathrm{s}]$$

Note that the local keys are sorted only once.

The split&merge operation. The parallel sorting algorithms discussed in this paper may all be described as sorting circuits. Such a circuit consists of a set of wires, each holding a single key, and a set of comparators, each connecting two wires and performing a compare/exchange operation on the corresponding keys. In our environment, each PE simulates a single wire, but as it holds more than just one key, we replaced the original compare/exchange operation by a split&merge (S&M) operation. With this operation, the PE that has to receive the minimum (P_{MIN}) gets the lower n keys, the other one (P_{MAX}) gets the upper n keys. Our implementation of the S&M operation works as follows (the description is given for PE P_{MIN}; the S&M operation works analogously for PE P_{MAX} starting with the uppermost keys): At first P_{MIN} fetches the lowermost key from P_{MAX} and compares it with the lowermost one from its own current sequence. The minimum is stored in the local resulting sequence. If the minimum comes from P_{MAX} then P_{MIN} fetches the next key from P_{MAX}, otherwise the next local key is chosen for the following comparison. After n steps, P_{MIN} (P_{MAX}) holds the n lower (upper) keys from P_{MIN} and P_{MAX}.

Since our algorithms are oblivious, after a given number of S&M operations (depending on the particular algorithm) the whole input sequence is sorted. However, it might be the case that depending on the input keys, the execution of some S&M operations becomes unnecessary, because the maximum key of P_{MIN} is already smaller than the minimum key of P_{MAX}. In such a case, the execution of the S&M operation can be avoided. We achieve this by inserting an additional comparison after receiving the first key and applying a plural if-statement [Mas93a] on the result of the comparison to determine whether the condition mentioned is true or not. We say that the S&M operation is *guarded* .

If we disregard the communication operations for the moment (the time to communicate a key may not be constant during the execution of an algorithm and varies from algorithm to algorithm because of different communication patterns; therefore we shall discuss this point later in connection with the sorting algorithms), one step of the S&M operation consists of loading two operands into registers, comparing them, and storing one of them back into the memory. Considering the overhead for address calculations, loop control, function call, and the additional comparison, we get as time consumption of one S&M operation:

$$t_{S\&M}(n) \quad = \quad n \cdot 80.6 + 117.9 \quad [\mu\text{s}]$$

Note that the proposed implementation of the S&M operation may be replaced by another one which makes the overall algorithm an in-place sorting routine with no additonal copies of keys and which consists of three phases: a binary splitter search to determine which keys to exchange, exchanging the keys and storing them in-place in reverse order (such that the resulting sequences become bitonic), and finally using a sequential implementation of the Bitonic Merging network [Bat68] to locally merge the bitonic sequences to sorted ones. This would increase the time complexity of the sorting algorithms by a factor of $\log n$ but also would be a useful modification if there was no more local memory available in addition to the input sequences.

Routing. Besides the internal sorting and the S&M operation, some of the algorithms consist of additional routing phases during which the contents of the whole torus is permuted in a particular manner. These routing phases are implemented by using the Global

Router and an appropriate library function (namely, `ss_rsend`) of the MasPar Programming Language (MPL) [Mas93a, Mas93b]. The runtime of this function is on average:

$$t_{Routing}(p) \quad = \quad (\mathrm{sf}_R \cdot (p \cdot 1.6 + 11.4) + 22.1) \quad [\mu\mathrm{s}],$$

where p is the packet size in bytes and sf_R the sequencing factor of Router communication. In our environment, $p = 4n$, and for the sorting algorithms, we can assume $\mathrm{sf}_R = 16$. Thus, the time consumption of a routing phase is:

$$t_{Routing}(n) \quad = \quad n \cdot 102.4 + 204.5 \quad [\mu\mathrm{s}]$$

4 Algorithms and Experimental Results

In this section, we describe the implementations, runtime predictions, and experimental results of the parallel sorting algorithms. Note that we always use all PEs due to the torus architecture that does not allow to use only a portion of the machine in a straight forward fashion. Using a portion would result in using the Router instead of the X-Net.

Since all implementations are based on the same fundamental routines, a common formula for runtime predictions can be given:

$$\begin{aligned} T_{method}(N,P) \quad = \quad & t_{SeqSort}(N/P) + \\ & \sigma_{method}(P) \cdot t_{S\&M}(N/P) + \\ & c_{method}(N,P) + \\ & \tau_{method}(P) \cdot t_{Routing}(N/P) \end{aligned}$$

One has to interpret the arguments of this formula as follows: At first, the input sequences of the processors are sorted locally by applying the sequential sorting routine with time consumption $t_{SeqSort}(N/P)$. The main work of the algorithms is done by the S&M operation with time consumption $t_{S\&M}(N/P)$. How often this operation is executed, $\sigma_{method}(P)$, depends on the specific algorithm and on the number of processors. Remember that the communication which is necessary to execute the S&M operation has not been taken into consideration yet. Therefore, $c_{method}(N,P)$ is the communication time of all S&M operations. Obviously, this time depends on the algorithm and its specific communication patterns, the number of keys, and on the number of processors. We will come back to that point later in this section, when we describe the algorithms. Finally, the parallel algorithm may include some routing phases, each requiring $t_{Routing}(N/P)$ time. The number of routing phases, $\tau_{method}(P)$, varies from one algorithm to the other and may depend on the number of processors.

4.1 Bitonic Sort

Bitonic Sort (BS) is one of the most famous methods in the context of parallel sorting. The algorithm was introduced as a sorting circuit by Batcher in 1968 [Bat68] (see Figure 2 (a) for the circuit with $P = 8$). According to the depth of that circuit, we get $\sigma_{BS}(P) = \frac{1}{2}\log P(\log P + 1)$ as the number of S&M operations.

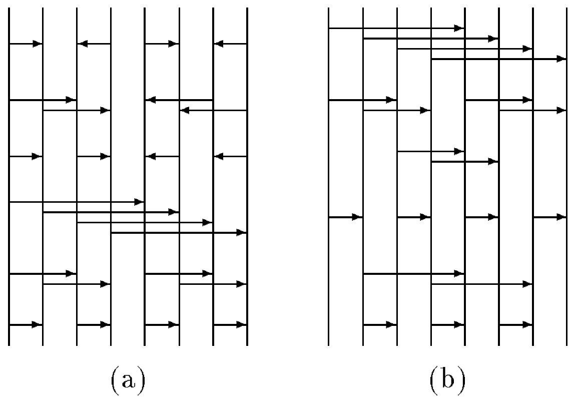


Figure 2: (a) Batcher's Bitonic Sort and (b) Odd-Even Merge Sort for input sequences of length 8.

A direct application of Bitonic Sort to a processor network requires the hypercube topology. Unfortunately, this is not the topology of the MasPar. Thus, we have to give an embedding of the hypercube of dimension $\log P$ ($Q_{\log P}$) into the two-dimensional mesh of size $2^{\lfloor \frac{\log P}{2} \rfloor} \times 2^{\lceil \frac{\log P}{2} \rceil}$. For that purpose, we choose the ascending balanced axes embedding (see Figure 3) which maps hypercube edges of low dimension onto short paths in the mesh. This is advantageous because low-dimensional edges are used more often than

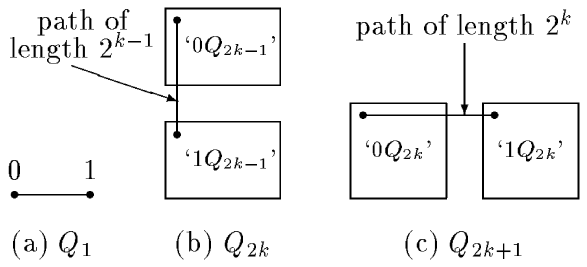


Figure 3: Ascending balanced axes embedding of the hypercube into the two-dimensional mesh.

high-dimensional ones. E. g., the edges of dimension 1 are used $\log P$ times, whereas those of dimension $\log P$ are only used once during the execution of the algorithm.

Now we are able to analyze the communication cost of Bitonic Sort. We are assuming that each execution of the S&M loop includes a communication operation which is carried out by using the X-Net. Furthermore, we are assuming that in general, communication takes place in both possible directions. Thus, $\mathrm{sf}_X = 2$ in most cases. However, for communication along the hypercube dimensions $\log P$ and $\log P - 1$ we use the wrap-around edges of the MasPar topology such that only one direction is used. Thus, in those cases $\mathrm{sf}_X = 1$. Altogether, we get as communication cost of Bitonic Sort:

$$c_{BS}(N,P) = \frac{N}{P} \cdot \left(\sum_{i=0}^{\log P - 3} (\log P - i) \cdot t_X(2, 2^{\lfloor i/2 \rfloor}) + \sum_{i=\log P-2}^{\log P - 1} (\log P - i) \cdot t_X(1, 2^{\lfloor i/2 \rfloor}) \right)$$

Finally, we make use of one routing phase at the end of the algorithm to rearrange the contents of the mesh according to the row major indexing scheme of the PE array. Thus, $\tau_{BS}(P) = 1$.

Measured and predicted times of Bitonic Sort are shown in Figure 4. As we used the original source code of Prins' implementation, all times could be measured on the same machine. The diagram shows the time that is spent by the algorithms per single key (time per key per PE), a common way to describe the performace of parallel sorting algorithms. For performance measurements, we made 1000 experiments with 2^i keys per processor (N/P), for each $1 \leq i \leq 10$. The corresponding curve represents the mean values of these experiments. There are very small deviations between the two curves. So our model and the worst case estimation of Bitonic Sort seem to be suitable for application to this algorithm. However, Figure 4 also shows that our simple approach for adapting Bitonic Sort is not able to outperform Prins' implementation .

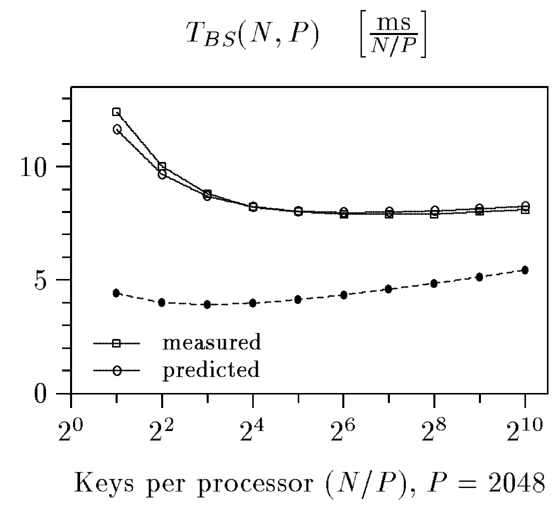


Figure 4: Predicted and measured times of Bitonic Sort. The dashed curve shows the times of the sophisticated implementation of Bitonic Sort by Prins [Pri90].

4.2 Odd-Even Merge Sort

Odd-Even Merge Sort (OEMS) was introduced as a sorting circuit in the above-mentioned paper by Batcher in 1968 [Bat68] (see Figure 2 (b) for the sorting circuit with $P = 8$). The abstract algorithm for Odd-Even Merge Sort is as follows:

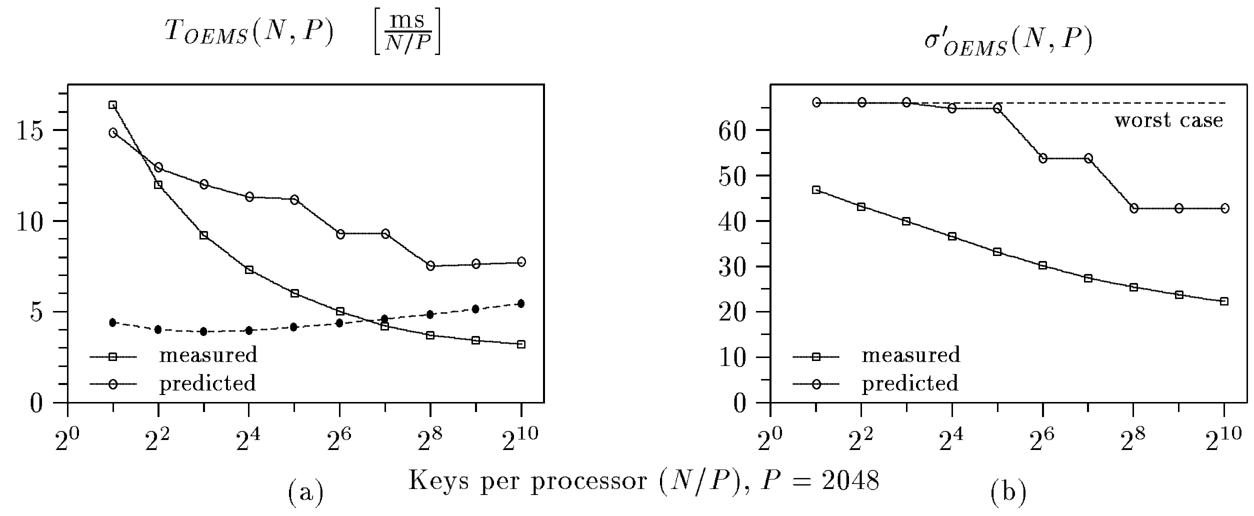


Figure 5: Odd-Even Merge sort: (a) predicted and measured times; (b) number of S&M operations executed. (The dashed line shows the times of Prins' implementation of Bitonic Sort.)

```
for all pid $\in \{0, \ldots, P-1\}$ do in parallel
    $\delta := P$;
    for $t := 0$ to $\log P$ do
        $\delta := \delta/2$;
        if even(pid div $\delta$) then S&M(pid, pid div $\delta$)
        for $i := 1$ to $\log(P/\delta)$ do
            if odd(pid div $\delta$) then S&M(pid, pid + $P/2^i - \delta$)
        od
    od
od
```

Of course, the depth of the corresponding circuit is the same as that of Bitonic Sort. Thus, $\sigma_{OEMS}(P) = \frac{1}{2}\log P(\log P + 1)$ in the worst case. Nevertheless, this algorithm was widely passed over in the context of parallel sorting on massively parallel processor networks because its communication structure is not as regular as that of Bitonic Sort.

In [Rüb95], Rüb investigates the average behavior of Odd-Even Merge Sort for large values of N/P. She proves the phenomenon that nothing happens during most of the S&M operations, i. e., the state of the network remains unchanged. If there are keys that have to be exchanged for any pair P_{MIN}, P_{MAX}, we call the S&M operation *active*; otherwise it is considered to be *non-active*. Rüb proves that the average number of active S&M operations is at most

$$\sigma_{active}(N,P) = \log P \left(1.89 + \left\lceil \log\left(1 + \sqrt{1.08P^2/N}\right)\right\rceil\right).$$

In our implementation, execution of the non-active S&M operation is avoided automatically (see Section 3). Thus, we can estimate the number of executed S&M operations with $\sigma'_{OEMS}(N,P) := \min(\sigma_{OEMS}(P), \sigma_{active}(N,P))$, i. e., the non-active S&M operations are ignored for runtime prediction.

To analyze the communication cost of Odd-Even Merge Sort, we assume that each execution of the S&M loop includes a communication operation. In our implementation, we use both X-Net as well as Router communication for that purpose. The X-Net is used when communicating processors lie on a horizontal or vertical line; otherwise the Router is chosen. Thus, we can roughly estimate the cost of one communication instruction as the mean value of t_X with $\mathrm{sf}_X = 2$ and $dist = 8$, i. e., both possible directions of the X-Net are used and the average distance between the communicating processors being eight, and t_R with $\mathrm{sf}_R = 8$, i. e., on average half of the PEs of all clusters synchronously send and/or receive a key. Thus, we get as communication cost of Odd-Even Merge Sort:

$$c_{OEMS} \quad = \quad \sigma'_{OEMS}(N,P) \cdot \frac{N}{P} \cdot \frac{t_X(2,8) + t_R(8)}{2}$$

Measured and predicted times of Odd-Even Merge Sort are shown in Figure 5 (a). The measurements are made under the same conditions as in the case of Bitonic Sort. There are significant deviations between the two curves which are due to the following facts: (1) Rübs theoretical analysis is not sufficiently tight, i. e., the experiments suggest that the number of executed S&M operations is less than stated above (see Figure 5 (b)). (2) The assumption that each execution of the S&M loop includes a communication operation may be reasonable for small values of N/P, but it is too pessimistic when N/P gets larger. In fact, we found out that with large N/P approximately only four out of five possible communication instructions are executed. (3) The estimation of the communication parameters is inaccurate. Especially, sf_R may vary significantly with different values of N/P.

However, all that should not detract from the fact that Odd-Even Merge Sort, together with the *guarded* version of split&merge, is a very efficient parallel sorting algorithm. In the experiments, it is faster than our implementation of Bitonic Sort, when N/P is larger than eight. It is even faster than the sophisticated implementation of Bitonic Sort by Prins [Pri90], when N/P gets large enough (break-even point: $N/P \approx 100$).

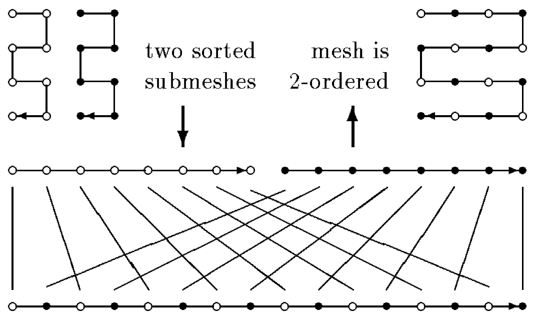
4.3 FastSort

FastSort (FS) is introduced in [Wan94]. It is a recursive algorithm which sorts meshes of arbitrary dimension according to the *snake-like* indexing scheme. The algorithm is based on Odd-Even Transposition Sort and makes use of some additional routing phases during which the contents of the mesh is rearranged in a specific manner. The description of FastSort is given for meshes of arbitrary dimension but with uniform size in each dimension which is supposed to be a power of two ($M(d,m)$, $m = 2^k$). It can easily be modified for our purposes when we have meshes that do not have equal side lenghts. The abstract algorithm is as follows:

```
procedure FastSort( $d$, $m$ )
  if $m = 2$ then
    sort the hypercubes of dimension $d$ by Bitonic Sort
  else
    sort all disjunct $(\frac{1}{2}m \times \cdots \times \frac{1}{2}m)$-meshes by
    applying FastSort( $d$, $\frac{1}{2}m$ )
    for $k := d$ downto 1 do
      { the $(\underbrace{m \times \cdots \times m}_{d-k} \times \underbrace{\frac{1}{2}m \times \cdots \times \frac{1}{2}m}_{k})$-meshes are already sorted }
      merge the pairs of $(\underbrace{m \times \cdots \times m}_{d-k} \times \underbrace{\frac{1}{2}m \times \cdots \times \frac{1}{2}m}_{k})$-meshes which
      are neighboring in dimension $k$
    od
  fi
end
```

To sort the hypercubes of dimension d, we may apply Bitonic Sort which needs $\frac{1}{2}d(d+1)$ parallel steps. Now we show how two sorted sub-meshes can be merged.

A sequence $(a_0, \ldots, a_{k-1})$ is called *2-ordered* if $a_i \leq a_{i+2}$, for all $0 \leq i < k-2$. It is shown in [Wan94] that if a mesh is 2-ordered according to the snake-like indexing scheme, it suffices to apply a descending run of Odd-Even Transposition Sort on the arrays in all dimensions. Thus, routing the snakes of the two sub-meshes according to the shuffle permutation establishes the desired state (see Figure 6). It is shown in [Wan94] that the



routing according to the shuffle permutation

Figure 6: Routing scheme of FastSort.

number of required S&M operations altogether is $\frac{1}{2}md(3d+1) - \frac{1}{2}d(5d+1)$.

To apply FastSort to meshes of dimension $d > 2$, an embedding of the d-dimensional mesh into the 2-dimensional PE-array of the MasPar is required. For that purpose, we choose the descending balanced axes embedding (see Figure 7). If, when embedding the arrays of dimension 1, the size of the current dimension of the PE-array is too short, we can 'fold' these arrays and use the wrap around edges of the X-Net to connect all nodes in a uniform manner (see Figure 8).

We have applied FastSort to four meshes of different dimensions. All relevant parameters are shown in Table 1. $\sigma_{FS,d}(P)$ is the exact number of S&M operations of our implemen-

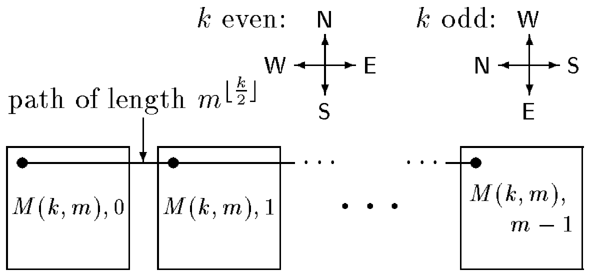


Figure 7: Recursion scheme of the descending balanced axes embedding of the $(k+1)$-dimensional mesh into the 2-dimensional one $(k \geq 2)$.

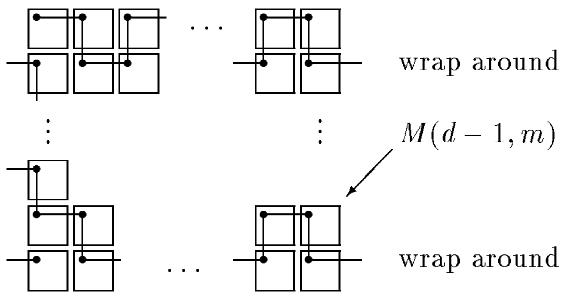


Figure 8: Special case of the descending balanced axes embedding for $d = 1$.

tation in the worst case. $dist_d$ is the mean distance of the communicating processors when executing the S&M operation. We estimate communication cost of FastSort by the worst case and get:

$$c_{FS,d}(N,P) \quad = \quad \sigma_{FS,d}(P) \cdot \frac{N}{P} \cdot t_X(2, dist_d)$$

The number of routing phases required, which is $\tau_{FS,d}(P) = \log P - d + 1$, includes a final routing phase to rearrange the contents of the mesh according to the row major indexing scheme of the MasPar.

Measured and predicted times of FastSort with the given parameters are shown in Figure 9. The performance measurements are made under the same conditions as in the case of Bitonic Sort and Odd-Even Merge Sort. There are significant deviations between predicted and measured times which are due to the following facts: First, the experiments suggest that the worst case estimation of $\sigma_{FS,d}(P)$ is too pessimistic (see Figure 10). This drawback

Table 1: The different parameters of FastSort.

d	geometry	$\sigma_{FS,d}(P)$	$dist_d$	$\tau_{FS,d}(P)$
2	64×32	216	1	10
3	$16 \times 16 \times 8$	131	6	9
4	$8 \times 8 \times 8 \times 4$	108	4.5	8
6	$4 \times 4 \times 4 \times 4 \times 2$	86	7	6

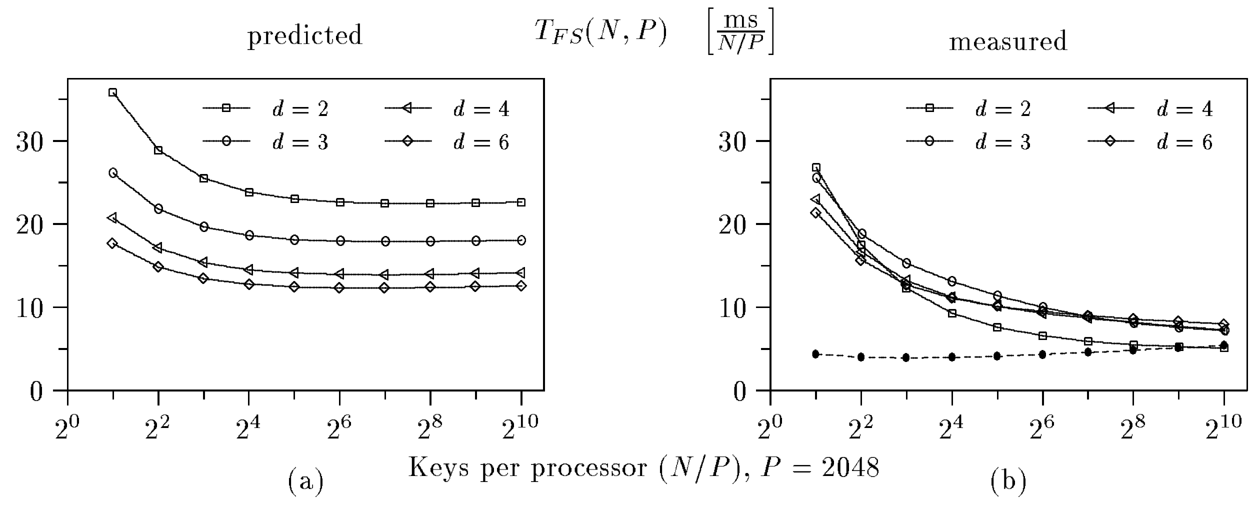


Figure 9: Predicted and measured times of FastSort.

is particularly significant when the edge size of the mesh is large. Another inaccuracy is the assumption that each execution of the S&M loop includes a communication operation, especially when N/P gets larger. In fact we found out that with large N/P approximately only two out of three possible communication instructions are executed. That there is a need for a strong average case analysis for this algorithm is demonstrated by the similarities of Figure 9(b) and Figure 10.

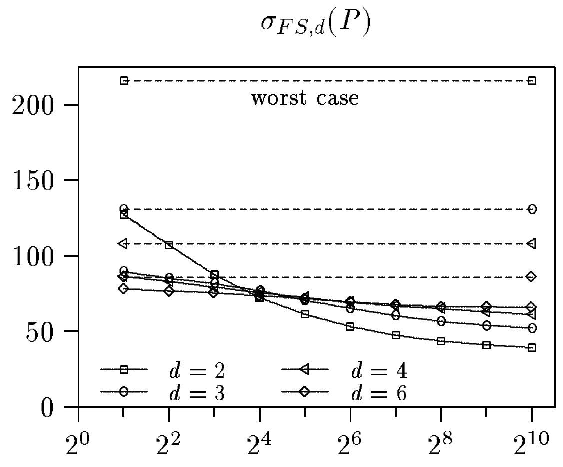


Figure 10: Number of executed S&M operations of FastSort – worst case and experimental results.

5 Conclusions

In this paper, we showed that, by using the guarded split&merge operation, from a certain ratio N/P upwards, Odd-Even Merge Sort and FastSort outperform a sophisticated implementation of Bitonic Sort. See Figure 11 for a comparison of the measued runtimes of our and Prins' implementations. This leads to the recommendation to use Odd-Even Merge

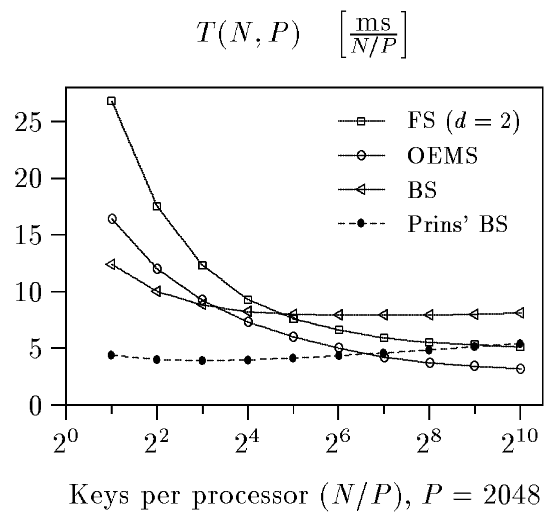


Figure 11: Measured runtimes of all algorithms.

Sort on small MP-1 systems whenever $N/P > 100$. Besides the need of more exact analyses of the average case behavior of Odd-Even Merge Sort and FastSort, it might be interesting to test our implementations on a MasPar MP-1 with more PEs and larger local memory, and to compare them to implementations of the randomized algoithms Samplesort and, in particular, B-Flashsort. Also it might be interesting to evaluate the performance of our implementations on the MasPar MP-2.

In the first author's Diplomarbeit [Bro95], implementations of further sorting circuits are treated: Shortperiodic methods, Shearsort and a combination of the Bitonic Merging network and Shearsort. This thesis as well as the source code of the algorithms can be downloaded from the WWW page http://wwwhni.uni-paderborn.de/cim/mitarb/brockm/ .

Acknowledgements

We would like to thank Brigitte Oesterdiekhoff for her valuable remarks and discussions about efficient sorting circuits, in particular about shortperiodic methods.

Christoph Eilinghoff and Friedrich Wichmann deserve our thank for their help when the sorting routines were included into an application library.

Last but not least, we are grateful to Mike Utermöhle for providing the technical support for our MP-1.

References

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS Conference Proc. 32*, pages 307–314, 1968.

[BLM+91] Guy E. Blelloch, Charles E. Leiserson, Bruce M. Maggs, C. Greg Plaxton, Stephen J. Smith, and Marco Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 3–16, 1991.

[Bro95] Klaus Brockmann. *Implementierung und Analyse paralleler Sortieralgorithmen auf einer MasPar MP-1 und ihre Einbindung in eine Anwendungsbibliothek.* Diplomarbeit, Paderborn University, 1995. In German.

[CK90] Bogdan S. Chlebus and Maciej Kukawka. A guide to sorting on the mesh-connected processor array. *Computers and Artificial Intelligence*, 9:599–610, 1990.

[DCSM96] Andrea C. Dusseau, David E. Culler, Klaus E. Schauser, and Richard P. Martin. Fast parallel sorting under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7:791–805, 1996. Preliminary version: Fast Parallel Sorting under LogP: From theory to practice. In *Portability and Performance for Parallel Processing*, pp. 71–98, 1994.

[DGL+94] Ralf Diekmann, Joern Gehring, Reinhard Lüling, Burkhard Monien, Markus Nübel, and Rolf Wanka. Sorting large data sets on a massively parallel system. In *Proc. 6th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 2–9, 1994.

[FM70] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *Journal of the ACM*, 17:496–507, 1970.

[HPR92] William L. Hightower, Jan F. Prins, and John H. Reif. Implementation of randomized sorting on large parallel machines. In *Proc. 4th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 158–167, 1992.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

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

[Mas93a] MasPar Computer Corporation, Sunnyvale, California. *MasPar Parallel Application Language (MPL) Reference Manual*, May 1993.

[Mas93b] MasPar Computer Corporation, Sunnyvale, California. *MasPar Parallel Application Language (MPL) User Guide*, May 1993.

[Pri90] Jan F. Prins. Efficient bitonic sorting of large arrays on the MasPar MP-1. In *Proc. 3rd Symposium on Frontiers of Massively Parallel Processing*, pages 59–64, 1990. Expanded version: Technical Report TR91-041, University of North Carolina at Chapel Hill, Dept. of Computer Science, 1991.

[Ric86] Dana Richards. Parallel sorting – A bibliography. *SIGACT News*, 18(1):28–48, 1986.

[Rüb95] Christine Rüb. On the average running time of odd-even merge sort. In *Proc. 12th Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 491–502, 1995.

[Wan94] Rolf Wanka. *Paralleles Sortieren auf mehrdimensionalen Gittern.* Ph. D. Thesis, Paderborn University, 1994. In German.

[WW96] Alf Wachsmann and Rolf Wanka. Sorting on a massively parallel system using a library of basic primitives: Modeling and experimental results. Technical Report TR-RSFB-96-011, Paderborn University, May 1996.

[ZCZ96] S.Q. Zheng, B. Calidas, and Yanjung Zhang. Efficient in-place sorting algorithms using feasible parallel machine models. In *Proc. 2nd International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, pages 15–21, 1996.