INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Sorting on a Massively Parallel System Using a Library of Basic Primitives: Modeling and Experimental Results*

Alf Wachsmann† **Rolf Wanka**‡

TR-97-028

August 1997

Abstract

We present a comparative study of implementations of the following sorting algorithms on the Parsytec SC320 reconfigurable, asynchronous, massively parallel MIMD machine: Bitonic Sort, Odd-Even Merge Sort without and with guarded split&merge, Periodic Balanced Sort, Columnsort, and two variants of Samplesort. The experiments are performed on 2- up to 5-dimensional wrapped butterfly networks with 8 up to 160 processors. We make use of library functions that provide primitives for global variables and synchronization, and we show that it is possible to implement efficient and portable programs easily. We assume the time for accessing a global variable to be linear in the parameters s, d, and c, where s is the size of the variable, d the distance between the accessing processor and the processor holding the variable, and c the contention, i.e., the number of processors accessing the variable simultaneously. In order to predict the performance, we model the runtime of this access by a trilinear function. Similarly, the runtime of a synchronization is described by a bilinear function, depending on the number of processors involved and their maximum distance. Our experiments show that, in the context of parallel sorting, this model that can be applied easily is sufficiently detailed to give good runtime predictions. The experiments confirming the predictions point out that Odd-Even Merge Sort with guarded split&merge is the fastest method if the processors hold few keys. If there are many keys per processor, a variant of Samplesort that uses Odd-Even Merge Sort as a subroutine is the fastest method. Additionally, we show that the relative behavior of implementations of different algorithms is quite similar to their theoretical relation.

*This research was conducted while both authors were affiliated with the Dept. of Mathematics and Computer Science and the Heinz Nixdorf Institute of the Paderborn University, Germany. It is supported by DFG-Sonderforschungsbereich 376 "Massive Parallelität," by EU ESPRIT Long Term Research Project 20244 (ALCOM-IT), and by DFG Leibniz Grant Me 872/6-1. A shortened version has been accepted for presentation on the 3rd European Conference in Parallel Processing (Euro-Par), 1997.

†DESY-IfH, D-15738 Zeuthen, Germany. e-mail: `alf@ifh.de`

‡ICSI, 1947 Center Street, Berkeley, CA 94704, USA. e-mail: `wanka@icsi.berkeley.edu`

1 Introduction

The problem. Sorting is one of the most investigated problems in computer science. In the area of parallel computing, sorting is a classical topic as well. Its roots can be traced back to the Fifties [19, p. 244]. Richards' bibliography [23] covers the vast 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. Important aspects in the design of parallel sorting algorithms are, among others, the used interconnection network of processors, the number and the size of keys per processor, and the desired output order. These aspects are treated, e. g., in Leighton's book [20] in detail. At the same time, parallel sorting is also a popular benchmark for parallel machines, because there are both algorithms that have a regular communication pattern and algorithms that communicate data in an irregular pattern. Therefore, it is considered a prototypical data movement task.

In contrast to sequential computing, where the Random Access Machine (RAM) model proved to be very successful, there are specific problems in designing models for parallel computing that are sufficiently detailed to give good runtime predictions and that can be applied easily. The reason is that, e. g., aspects of inter-processor communication influence the runtime in a way that cannot be handled easily because they depend on many hardware and algorithmic parameters. The more exact a mathematical description is, the more difficult it is to survey and to apply. Often, not even the relative behavior of parallel algorithms that is determined by theoretical analysis can be observed when the algorithms are implemented.

Previous work. Experimental comparative studies on a variety of different parallel sorting algorithms are presented by Blelloch *et al.* [6], by Culler *et al.* [10, 13], and by Diekmann *et al.* [11]. The first study gives a detailed discussion of implementations on the 64K-processing elements SIMD machine CM-2, interconnected as an 11-dimensional hypercube. In the second study, it is reported on implementations on the MIMD machine CM-5, interconnected as a 32- to 512-node fat tree, and, in the third, implementations on the MIMD machine Parsytec GCel with 32×32-mesh architecture are presented. These investigations turn out that up to a certain number k of keys per processor, Batcher's Bitonic Sort [4] is the fastest method. If more than k keys are stored per processor, Samplesort [15] is the fastest. Batcher's Odd-Even Merge Sort [4] has not been considered in these studies.

Single methods' implementations that have influenced our work are described in Stricker's paper [27] and in Rüb's paper [24, 25]. Stricker implemented and analyzed Bitonic Sort on an iWarp torus, Rüb describes and proves a phenomenon significantly speeding up Odd-Even Merge Sort on average when the number of keys per processor is large. She has implemented her method that is based on guarded split&merges on an Intel iPSC/860.

In a recent study [7], we implemented a variety of circuit-based sorting methods on a SIMD machine MasPar MP-1 of moderate size. The implementation of Rüb's variant of Odd-Even Merge Sort proved to be very effienct.

In [26], implementations of Bitonic Sort and Samplesort on the NEC Cenju-3 and the IBM SP2 are discussed.

In the context of the search for a theoretical model of parallel computing, there are two models which are considered well suited for describing the behavior of parallel algorithms on real machines. For the LogP model [9], Culler *et al.* [10] compare predictions and measurements of implementations of parallel sorting algorithms. For the BSP model [28], Gerbessiotis

and Valiant [17] prove that Samplesort is 1-optimal for a certain range of the parameters of the BSP model. 1-optimal means that a p-processor parallel algorithm is asymptotically p times faster than the fastest sequential algorithm, and that the time for communication is asymptotically smaller than that for computation. Goodrich [18] provides a BSP sorting algorithm that is 1-optimal for all p.

Contribution of this paper. We present a comparative study of implementations of the following sorting algorithms on the reconfigurable, asynchronous, massively parallel MIMD machine Parsytec SC320 [16] consisting of 320 INMOS Transputer processors: We have implemented Batcher's sorting circuits Bitonic Sort and Odd-Even Merge Sort [4], Rüb's version of Odd-Even Merge Sort with guarded split& merge [24, 25], the periodic balanced sorting circuit [12], Leighton's Columnsort [20, 21], and two variants of Samplesort [15]. In the first variant of Samplesort, the set of all samples is sorted sequentially in one processor, in the second variant, this set is sorted in parallel by Rüb's version of Odd-Even Merge Sort. The experiments are performed on the SC320 interconnected as a 2-dimensional up to a 5-dimensional wrapped butterfly network, i.e., on 8 up to 160 processors. These implementations are tested with inputs generated according to the NAS Integer Benchmark [2, 5, 22] collection. Note that we here focus on methods, which do not rely on the representation of the keys.

The algorithms are implemented in a language that makes it possible to write PRAM style programs which are therefore also portable. This is realized by modifying occam 2, the language of the Transputer processor, and introducing a library that realizes global variables and provides routines for synchronization. With this extended language, it is easy to write efficient programs. E.g., see Figure 1, where the code for the Periodic Balanced Sort is presented. The programs for the other comparator-based methods are similar. Also the combination of Rüb's variant of Odd-Even Merge Sort with Samplesort can be done easily. Because the programs do not contain occam 2 specific constructs, it is easy to directly translate them into, e.g., SPLIT-C, a widely used parallel language.

Additionally, the measurements turn out that the programs run efficiently, that the runtime behavior scales, and that the theoretical and measured relative behavior of the algorithms are quite similar.

Odd-Even Merge Sort which has not been considered in the previous comparative studies mentioned above is the fastest method for few keys per processors. If there are many keys per processor, the fastest algorithm is the second version of Samplesort. More specifically, in Figure 11 it is recommended in which situation (depending on network size and number of keys per processor) which algorithm should be chosen.

In order to give performance predictions, we use the MLF model [14, 29], where single portions in the overall runtime like routing time etc. are modeled by multilinear functions. Here, the time that is necessary to access a global variable is assumed to be linear in the size of the variable, the distance between the accessing processor and the processor holding the variable, and the contention, i.e., the number of processors accessing the variable simultaneously. Thus, we model the runtime of this access by a trilinear function. Similarly, the runtime of a synchronization is described by a bilinear function, depending on the number of involved processors and their maximum distance. Our experiments show that, in the context of parallel sorting, this model is sufficiently detailed to give good runtime predictions. Moreover, this model is sufficient to rate changes in the algorithm: E.g., the influence of the introduction of

Rüb's guarded split&merge operation in Odd-Even Merge Sort is clearly to recognize, as well as the speedup of combining Odd-Even Merge Sort and Samplesort.

We refrained from implementing Cole's Merge Sort [8] and, of course, the AKS sorting circuit [1] though these algorithms are theoretically optimal. The constant factors involved in their running times are so large that it is obvious that they cannot outperform the other algorithms. In contrast to Blelloch *et al.* [6], we did not implement Radixsort because this method relies on the representation of the keys.

Thus, the paper presents an application of a variety of methods for implementing and modeling that is also interesting considered independently of the used hardware.

The paper is organized as follows: In Section 2, we give a detailed description of the used MLF model. In Section 3, we describe the parts that are common to all implementations. In Section 4, we present the runtime predictions and measurements in detail.

2 Model and Experimental Platform

The MLF Model. Many existing parallel processor networks (e.g., the Cray T3D, the Thinking Machines CM-5, the Parsytec SC320, etc.) can be viewed as a set of powerful MIMD processors that are interconnected as a graph (e.g., a two-dimensional mesh, a fat tree, a butterfly network). Communication is performed in a wormhole or store-and-forward fashion. These communication methods lead to the hypothesis that the time for sending a message m from a processor p to a processor p' is linear in the size s of m and linear in the length d of the path from p to p'. If the same message m has to be sent from p simultaneously to c different destinations (without knowing more specific information), p can satisfy these requests only sequentially, i. e., in time linear in c. A similar argument holds when c messages are sent to the same global variable. Thus, the time $\mathrm{agv}(d, s, c)$ of simultaneously accessing a global variable, stored in processor p, by c different processors that all have distance at most d to p can be expressed by a multilinear function (MLF)

$$\mathrm{agv}(d, s, c) = \alpha_1 \cdot dsc + \alpha_2 \cdot ds + \alpha_3 \cdot dc + \alpha_4 \cdot sc + \alpha_5 \cdot d + \alpha_6 \cdot s + \alpha_7 \cdot c + \alpha_8 \ .$$

Note that this function is linear in each parameter. Also note that this function does not take into account how many requests have to be satisfied by a single processor. Only the simultaneous requests of the same variable are counted. The idea behind this approach is that it is possible to determine the number of simultaneous requests, but not the total number of requests that a single processor has to answer, because the user does not generally know where the variables are stored.

If we want to synchronize P processors that have a maximum distance d, and if we again assume that the time $\mathrm{sync}(d, P)$ for this task is linear in d and P, we have

$$\mathrm{sync}(d, P) = \beta_1 \cdot dP + \beta_2 \cdot d + \beta_3 P + \beta_4 \ .$$

In these formulas, the constants α_i and β_j have to be determined by a large number of experiments. Therefore, these formulas only approximate the real runtime. The real runtime can be faster or slower. This model is applicable to parallel systems with store-and-forward routers as well as to systems with wormhole routers. The mode of the router is covered by the measured constants. For systems where the distance d between the processors does not play an important role, the constants $\alpha_1, \alpha_2, \alpha_3, \alpha_5$, and β_1, β_2, respectively, should be small. In [14], this model is used for different interconnection networks and different routing strategies.

```
VAL INT No.Procs IS 160:              – number of processors in the network
VAL INT P IS 256:                     – power of 2 greater than No.Procs
VAL INT N.over.P IS 1024:             – number of keys per processor
[No.Procs] PUBLIC [N.over.P] INT32 AT 0 x: – global array for the keys
[N.over.P] INT32 Local.Keys, Partner.Keys: – for local copies of keys

SEQ
  init.data.nas(pid, Local.Keys)      – NAS Integer Sorting Benchmark
  qsort(Local.Keys)                   – sort local keys sequentially
  x[pid] := Local.Keys                – write local copy of keys to global variable

  SEQ t = 1 FOR log(p)                – for t := 1 to log P do
    SEQ kh = 1 FOR log(p)
      VAL INT k IS (log(p) - kh):     – for k := log P downto 1 do
      VAL INT partner IS (pid EXOR ((2<<k)-1)):
      SEQ
        SYNC(0, No.Procs)             – synchronization of all processors
        IF                            – if pid div 2^k is even and if pid ⊕ (2^(k+1) - 1) exists
          even(pid >> k) AND (partner < No.Procs)
            SEQ                       – execute an S&M op. between x[pid] and x[partner]
              PAR                     – latency hiding
                Local.Keys   := x[pid]
                Partner.Keys := x[partner]                              (*)
              Split.And.Merge(Local.Keys, Partner.Keys)
              PAR
                x[pid]     := Local.Keys
                x[partner] := Partner.Keys                              (*)
          TRUE
            SKIP
```

Figure 1: OCCAM-light code for the Periodic Balanced Sort. Accesses to global variables are marked with ($\star$).

Because of the good runtime predictions, the advantage of this model is that it is quite easy to rate in advance changes of an algorithm, e. g., if a certain procedure is replaced by another one. In this paper, this is is used to rate the modification of Odd-Even Merge Sort (Section 4.1.2) and of Samplesort (Section 4.3). If there is a good theoretical understanding of the implemented methods, also the predictions are reasonably good.

The drawback of this model is the following. The measured constants α_i and β_j depend on the used parallel system and on the implementation of the basic primitives. Once they are determined, it is hardly possible to detect the influence of hardware-related parameters as, e. g., the start-up times, or of a change of the implemented routing strategy. So, this model can be used to decide which algorithms should be implemented on a given, experimentally evaluated platform. It cannot be used to find out which parameters in a parallel system should be improved.

The Experimental Platform. occam 2 is the language of the INMOS Transputer processor. It has primitives for neighbor-to-neighbor communication. There are the following problems, when one wants to implement specific algorithms.

The communication pattern is different from the graph of the processor network. Thus,

information between neighboring processes has to be routed through the network. Therefore, the programmer has also to implement this routing, using the primitives for neighbor-to-neighbor communication.

Often, processes need the same information simultaneously. The programmer has to place it on a predefined location and has to implement routing phases to access this shared information.

Usually, parallel algorithms, in particular sorting algorithms, are designed for a *synchronized* setting. If they are implemented on an asynchronous machine without forced synchronizations, they do not work correctly. Therefore, there is the need for a mechanism to let processes wait, until all or certain processes have reached a specific point of their computation to proceed.

Once the software has been implemented, it cannot be used for another network for it depends on the network because of the hand-made routing subroutines of the program.

In order to overcome these drawbacks, occam 2 has been extended by primitives for routing, shared global memory, and synchronization [30, 29]. This language is called OCCAM-light. It leads to a situation similar to the situation on sequential machines: The basic routines are no longer in the responsibility of the programmer, because they are implemented with the help of special knowledge and efficiently. E. g., very large packets to be sent are split into packets of a size that is optimized with respect to the throughput of the system so that a kind of wormhole routing is realized.

Thus, it is possible to write portable software. OCCAM-light enables the programmer to write PRAM-like programs. See the code in Figure 1 for an OCCAM-light program which shows the entire source of the Periodic Balanced Sort. Because the programs do not contain occam 2 specific constructs, it is easy to translate them directly into, e. g., SPLIT-C, a widely used parallel language.

OCCAM-light has been implemented on the SC320 machine [16], interconnected as a k-dimensional wrapped butterfly network consisting of $k \cdot 2^k$ processors, $k \in \{2, \ldots, 5\}$. The routing is realized by routing on a shortest directed path (which is not in general a shortest path in the undirected graph).

Running this system with many different sample routing and synchronization problems, we measured the constants α_i and β_j shown in Table 1(a) and (b), respectively.

$\alpha_1 = 0.202$	$\alpha_2 = 7.445$	$\alpha_3 = 20.207$
$\alpha_4 = 0.509$	$\alpha_5 = 468.212$	$\alpha_6 = -12.259$
$\alpha_7 = 13.809$	$\alpha_8 = -223.388$	

(a) $\mathrm{agv}(d, s, c)$.

$\beta_1 = -8.921$	$\beta_2 = 325.069$
$\beta_3 = 92.331$	$\beta_4 = 261.199$

(b) $\mathrm{sync}(d, P)$.

Table 1: System-dependent coefficients for the multilinear functions (Time in μs).

The time GlobVis(s) to copy a local variable of size s to a global variable on the same processor is given by

$$\mathrm{GlobVis}(s) = 2.359 \cdot s + 229.663 \quad [\mu\mathrm{s}] \ .$$

3 Common Properties of the Used Sorting Methods

Let N be a multiple of P. We have P processors and N keys that are initially distributed among the P processors evenly, i. e., each processor holds N/P keys. In a single processor, the keys are stored in a linear array so that the time to access an arbitrary position in the array does not depend on the position. All of our implementations share some further common details we list in the following.

Access to Global Variables and Synchronization. The diameter of the k-dimensional wrapped butterfly network is $k^* = 2k - 1$, the length of the shortest path between two processors is $\bar{k} = \frac{3}{2}k + \frac{1}{2} - \frac{3}{k} + \frac{3}{k \cdot 2^k}$ on average. As we do not optimize the distribution of the global variables, we assume that accesses are always performed over a distance of k^*. Note that OCCAM-light provides the possibility to place the variables "by hand". Thus it is sometimes possible to speed up the running times by carefully investigating the algorithms and adapting them for the wrapped butterfly network.

In all synchronizations, all processors are involved, so we have to use k^* in the formula for the synchronization.

Internal Sorting: Quicksort. The algorithms have phases during which the keys stored in a single processor have to be sorted. For this *internal sorting*, we use the the well-known Quicksort algorithm in its variant 'best of three'. This algorithm works as follows: Three keys are chosen randomly from the set of keys. Their median is used as the *splitter* element. The set of keys is partitioned into the sequence of keys that are smaller than the splitter and into the sequence of keys that are larger than or equal to the splitter. The two sequences obtained in this way are sorted recursively. The expected number of comparisons this algorithm needs to sort n keys is $1.39n \log n$. Incorporating this into the cost model, we get as time consumption of Quicksort: $t_{\text{qsort}}(n) = 5.5 \cdot n \cdot \log n + 1300$ [μs].

Comparator Circuits and the Split&Merge Operation. With the exception of Columnsort and Samplesort, all implemented algorithms are directly based on sorting circuits. In a sorting circuit, each parallel step consists of an execution of a prespecified set of comparators. A comparator [a:b] connects two wires a and b each holding a key and performs a compare/exchange operation, i. e., the maximum is sent to b and the minimum to a. The number of parallel steps is the *depth* of the circuit. In our implementations, a processor simulates a single wire. As each processor holds more than one key (in fact, each processor holds a large number of keys), we replace each original compare/exchange operation by a split&merge (S&M) operation [19, p. 241]. I. e., the processor that has to receive the minimum gets the lower N/P keys, the other one gets the upper N/P keys. This operation is implemented as follows: In a Transputer network, it is quite expensive to transmit small packets consisting of a set of few keys from one processor to another. Therefore, it has no advantage to only determine the keys that have to be sent to the other processor. The processor U that has to receive the upper sequence sends all its keys to its companion which merges the two sorted sequences and then sends the upper sequence to U. We call the replacement of a parallel compare/exchange step by S&M operations an S&M *stage*. The internal runtime of an S&M operation on two sorted sequences of length n is on average: $t_{\text{S\&M}}(n) = 9.8 \cdot n$ [μs].

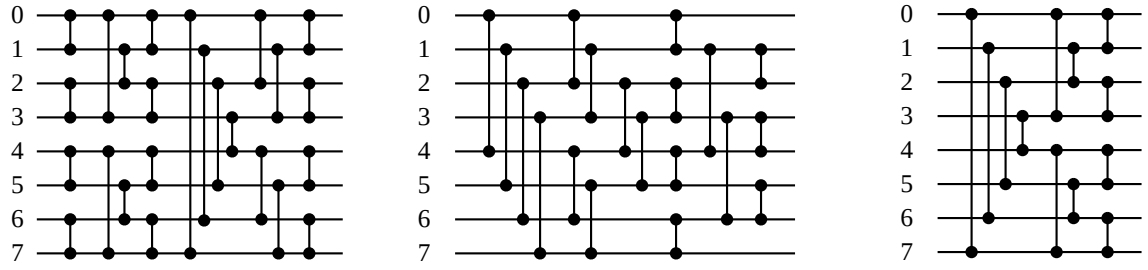
***P* Is Not a Power of 2.** Among others, we implement the *Odd-Even Merge Sorting* circuit, *Bitonic Sorting* circuit, and the *Periodic Balanced Sorting* circuit. These circuits are only defined for P being a power of 2. They only use comparators that are in standard form, i. e., the minimum is placed on the wire with smaller index. Therefore, we can apply the following observation. Let us have a sorting circuit for P keys with all comparators being in standard form. If we remove the last k wires and the comparators incident to them, the remaining circuit is a sorting circuit for $P - k$ keys. Note that this statement is not always true if some of the incident comparators are not in standard form. Fortunately, all sorting circuits can be transformed into an equivalent circuit consisting of standard comparators only [19, p. 239]. Thus, we implement the algorithms for $2^{\lceil \log P \rceil}$ wires and test before executing a comparator whether the maximum should be sent to a non-existing wire.

The NAS Parallel Sorting Benchmark. To evaluate the performance of our implementations, we apply them to sequences of 32-bit keys that are generated according to the IS (Integer Sorting) benchmark of the NAS Integer Benchmark [3] collection.

4 Runtime Predictions and Experimental Results

4.1 Circuit-Based Sorting Methods

In this subsection, we describe the implementations, runtime predictions and experimental results for three sorting circuits and a method that is based on a sorting circuit. See Figure 2 where the circuits are presented.



(a) Bitonic Sort. (b) Odd-Even Merge Sort. (c) A single block of the Periodic Balanced Sort.

Figure 2: Three comparator circuits with $P = 8$.

For sorting circuits without modifications, the formulas for the runtime predictions have a common structure. Let $\tau_{\text{method}}(P)$ be the depth of the sorting circuit 'method,' and let k be the dimension of the butterfly network. Thus, $P = k \cdot 2^k$, and the diameter is $k^* = 2k - 1$. Then the runtime prediction $T_{\text{method}}(N, P)$ is:

$$T_{\text{method}}(N,P) = t_{\text{qsort}}(\tfrac{N}{P}) + \text{GlobVis}(\tfrac{N}{P}) + \tau_{\text{method}}(P) \cdot \left(\text{sync}(k^*, P) + 2 \cdot \text{agv}(k^*, \tfrac{N}{P}, 1) + t_{\text{S\&M}}(\tfrac{N}{P})\right)$$

The arguments for this formula are as follows (see Figure 1): After the generation of the input (which is, of course, not counted for the runtime), the keys in each processor are sorted

by Quicksort locally. Then each processor `pid` writes its local keys to a variable that also is stored on `pid` and that is visible to all other processors. After this, $\tau_{\text{method}}(P)$ split&merge stages will be repeated. Before executing the tth S&M stage, it has to be ensured that all processors want to start the tth S&M operation. This is done by a synchronization. Omitting these synchronizations results in unsorted outputs. Now, each processor `pid` tests whether it has to take part in this S&M stage (if P is not a power of 2, see above). An S&M operation is realized by a simultaneous read from a local and a global variable. Then the actual S&M operation is executed. Now, the results of this operation are written to a local and a global variable. The runtime of the read/write phases is realized by latency hiding, i. e., local and global access are performed simultaneously, so that the runtime is dominated by the access to the global variable. Altogether, there are two accesses to global data, one reading access and one writing access, during a single S&M operation.

With the exception of the Odd-Even Merge Sort with guarded split&merge (Section 4.1.2), all circuit-based algorithms are described by the formula given above. Note that no contention occurs in the context of sorting circuits.

4.1.1 Bitonic Sort and Odd-Even Merge Sort

Both the bitonic sorting circuit and the odd-even merge sorting circuit have a depth of $\tau_{\text{bitonic/odd-even merge}}(P) = \frac{1}{2}\lceil \log P \rceil (\lceil \log P \rceil + 1)$. For the circuits with $P = 8$, see Figure 2 (a) and (b), respectively. Note that we have replaced the 'usual' bitonic sorting circuit by a variant consisting of comparators in standard form only. The abstract algorithm for Odd-Even Merge Sort is as follows. Note that here P has to be a power of 2 and therefore the test whether the communication partner exists is not necessary.

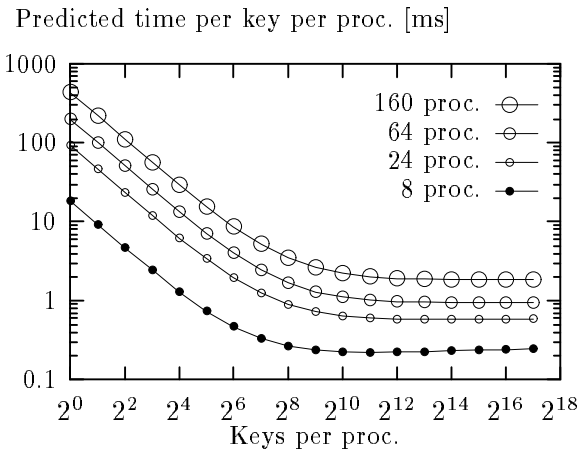
for all `pid` $\in \{0, \ldots, P-1\}$ **do in parallel**
  $\delta := P$;
  repeat $\log P$ **times**
    $\delta := \delta/2$;
    if even(`pid` **div** δ) **then** S&M(`pid`, `pid` $+ \delta$)
    for $i := 1$ **to** $\log(P/\delta) - 1$ **do**
      if odd(`pid` **div** δ) **and** (`pid` $+ P/2^i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

Figure 2(b) is a graphical representation of this algorithm with $P = 8$. This abstract algorithm can easily be coded in OCCAM-light.

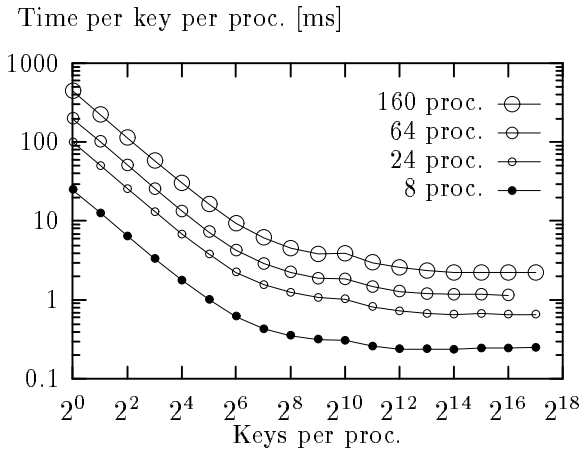
Figure 3(a) shows the predicted runtime for both algorithms, Figures 3(b) and (c) show the respective measured runtimes. It turns out that Odd-Even Merge Sort is slightly faster than Bitonic Sort, because the number of S&M operations of the first method is less than in the second. The total difference is $\frac{1}{2}P \log P + 1 - P$, and during a single time step, the difference can be up to $\frac{1}{4}P$, which results in better runtimes during the access to the global variables.

The bend that can be observed at 1024 keys per processor in the measured times is due to the fact that packets of this or larger size are split into smaller packets by the OCCAM-light library routines for routing. This leads to a kind of wormhole mode for large messages.

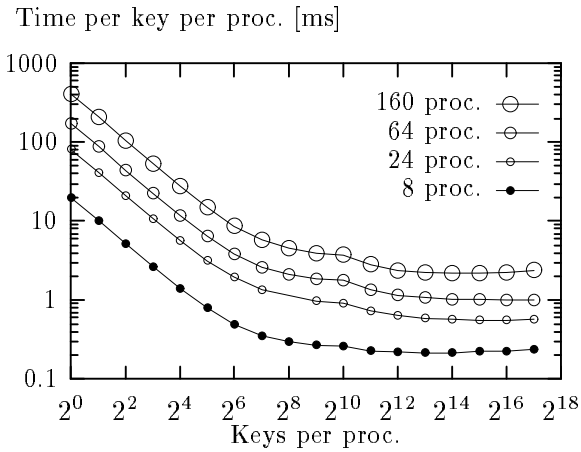
By using its hypercubic structure, Bitonic Sort could be adapted to a specialized implementation on the butterfly network. However, such an implementation is not hardware independent, and the speedup is not significant.



(a) Prediction

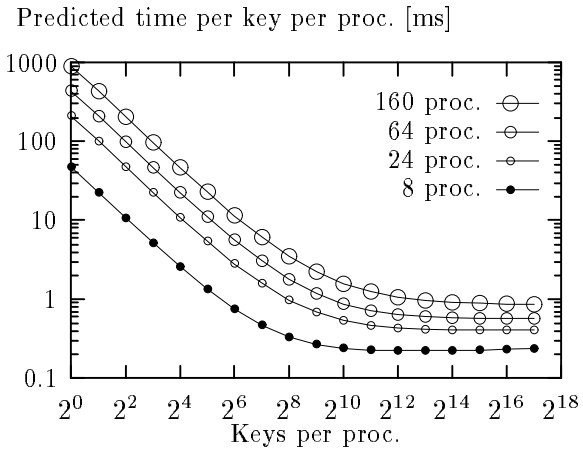


(b) Measurement for Bitonic Sort

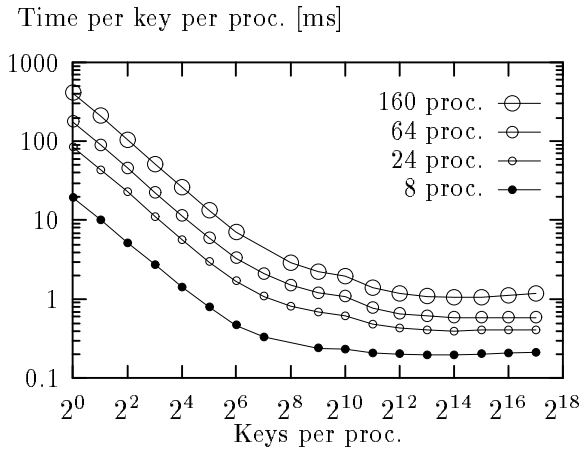


(c) Measurement for Odd-Even Merge Sort

Figure 3: Bitonic and Odd-Even Merge Sort.



(a) Prediction



(b) Measurement

Figure 4: Performance of Odd-Even Merge Sort with guarded split&merge.

4.1.2 Odd-Even Merge Sort with Guarded Split&Merge

In [24, 25], Rüb investigates the average behavior of Odd-Even Merge Sort with large N/P. She proves the following phenomenon that can reduce the runtime substantially: It can be expected that nothing happens during most of the S&M operations, because the keys that are treated by a single S&M operation are already in the right order. A parallel S&M stage is called *active*, if there are keys that have to be exchanged during this stage. Rüb proves that the average number of active S&M stages is at most

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

Therefore, we replace the usual S&M operation by the following *guarded* variant: The two processors that take part in a single S&M operation decide whether the S&M operation has to be executed as follows. The processor that has to receive the upper sequence sends its minimum key a to the processor L that has to receive the lower sequence. Only if a is smaller than the maximum key that is stored in L, the S&M operation is executed. Note that all stages are active in the worst case, and that introducing guarded S&M operations in Bitonic Sort results in an increase of the runtime. Considering the time necessary to send the small packets containing the minimum keys, we get the following prediction for the average runtime:

$$\begin{aligned} T_{\text{Rüb}}(N,P) = t_{\text{qsort}}(\tfrac{N}{P}) + \text{GlobVis}(\tfrac{N}{P}) + \tau_{\text{odd-even merge}}(P) \cdot \big(\text{sync}(k^*,P) + 2 \cdot \text{agv}(k^*,1,1)\big) \\ + \tau_{\text{active}}(P) \cdot \big(\text{sync}(k^*,P) + 2 \cdot \text{agv}(k^*,\tfrac{N}{P},1) + t_{\text{S\&M}}(\tfrac{N}{P})\big) \end{aligned}$$

If $\tau_{\text{active}} \ll \tau_{\text{odd-even merge}}$, then we obtain a considerable speedup.

Figure 4 presents the predicted and measured runtimes for Odd-Even Merge Sort with guarded split&merges. Compared to the original variant, this variant of Odd-Even Merge Sort saves about 50% in the runtime per key for large N/P. The deviation between the predictions and the measurements in this case is due to the fact that the theoretical analysis is not sufficiently tight, i. e., the experiments suggest that the number of active S&M stages is less than stated above.

Figure 5 shows that this algorithm mainly spends its time communicating global variables.

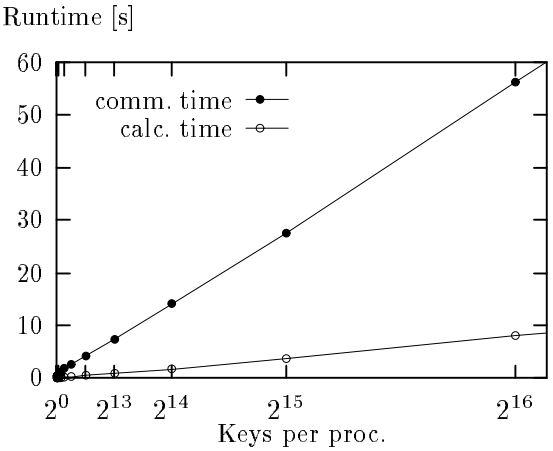


Figure 5: Time spent for calculation and communication in Odd-Even Merge Sort with guarded split&merge on the 5-dimensional butterfly network (cumulative).

This algorithm is used as a subroutine in the fastest variant of Samplesort in Section 4.3.

P	predicted	measured
8	1.5	1.4268
24	1.6667	1.6508
64	1.7142	1.7498
160	1.7778	1.7892

Table 2: Ratio of the runtimes of Periodic Balanced Sort and Bitonic Sort.

4.1.3 Periodic Balanced Sorting Circuit

The periodic balanced sorting circuit [12] realizes a *periodic* sorting method, i. e., there is a sequence of comparators (called a *block*) that is repeated until the input sequence is sorted. The block of the periodic balanced sorting circuit has a depth of $\lceil \log P \rceil$, and it has to be applied $\lceil \log P \rceil$ times in the worst case. Thus, its total depth is $\tau_{\mathrm{PBS}}(P) = \lceil \log P \rceil^2$. See Figure 2(c) for a single block with $P = 8$. We have implemented this algorithm in order to compare its worst-case behavior with the behavior of Bitonic Sort. That means our implementation applies the block exactly $\lceil \log P \rceil$ times, though the algorithm can be terminated if during an iteration no keys are moved.

Theoretically, the ratio of the runtimes is

$$\frac{\tau_{\mathrm{PBS}}(P)}{\tau_{\mathrm{bitonic}}(P)} = \frac{2}{1 + 1/\lceil \log P \rceil} \quad .$$

Table 2 shows the predicted and the measured ratios.

Beside the already described possible early termination, there also seems to be the effect that guarded S&M operations can speed up the algorithm on average. But nor the expected number of iterations of the block is yet known, neither a similar analysis as for Odd-Even Merge Sort.

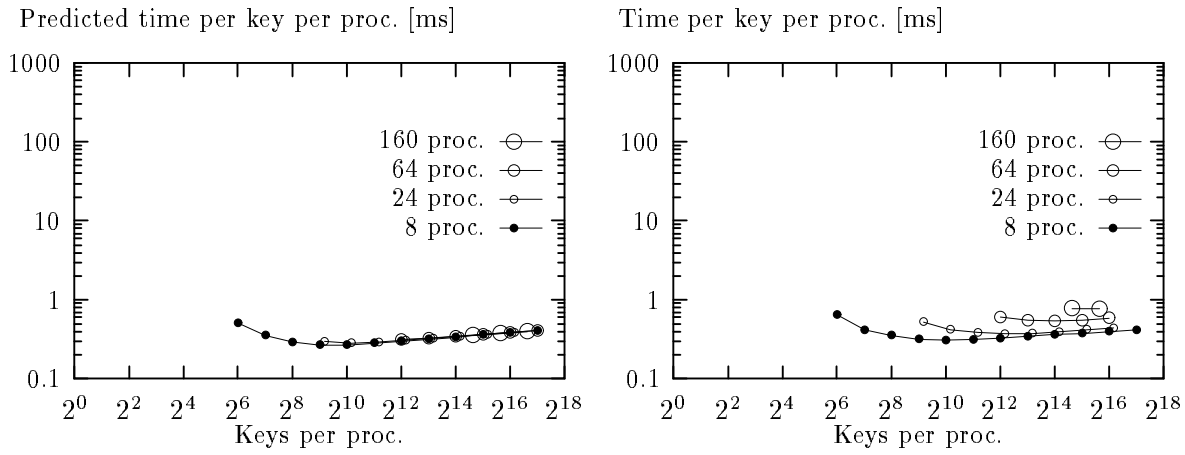
4.2 Columnsort

Columnsort has been introduced by Leighton in [21]. We have implemented the slightly modified version presented in [20, p. 261]. It is an algorithm for $m \times n$-meshes with $m \geq n^2$. It consists of four phases during which the columns of the mesh are sorted, of two phases during which only routing takes place, and one phase during which two compare/exchange steps are performed in the rows very similar to Odd-Even Transposition Sort [19, p. 241]. If sorting circuits are used to sort the columns, Columnsort reduces to a comparator-based method. In our implementation, a single processor simulates a column, i. e., $P = n$ and $N/P = m$. The sorting of the contents of the 'columns' is done by Quicksort. Thus, our algorithm reduces to a method for a linear array by this approach.

On the 5-dimensional butterfly network, this algorithm is slightly faster than the first version of Samplesort (see Section 4.3), if the number of keys is very large.

The predictions and measurement are given in Figure 6.

This method gives many possibilities for improvements. E. g., we implemented it with a placement of the variables according to a Hamiltonian path of the wrapped butterfly network, i. e., we used a numbering of the processors that allows neighbor-to-neighbor communication in the last phase. However, this does not speed up the method considerably (only the Odd-Even Transposition phase is realized faster than before), and the effort for the programming



(a) Prediction

(b) Measurement

Figure 6: Performance of Columnsort.

is significantly increased. Additionally, the efficiency of this implementation depends on the hardware. A further improvement can be realized by applying during the internal sorting in the higher phases sequential algorithms other than Quicksort, because Quicksort becomes slow when its input is well presorted. Also realizations are possible for the case that $m \leq n^2$.

As all these improvements are already outperformed by Samplesort for small N/P, they are not discussed here in detail.

4.3 Samplesort

Samplesort [15] is a generalization of Quicksort (see Section 3). For large networks and a large number of keys per processor, this method is usually the fastest, as mentioned in the introduction. Our aim in this section is to rate variations in the realization so that it is possible to determine their success in advance, i. e., before it has been implemented. Samplesort works in three phases:

- During Phase 1, each processor chooses randomly $b = \max\{r \cdot \frac{N}{P}, 1\}$ keys from its set of keys, $0 < r < 1$. The set of chosen keys is called *sample*. The sample is sorted, and then the $P-1$ elements with ranks $i \cdot \left\lfloor \frac{b}{P-1} \right\rfloor$, $i \in \{1, \ldots, P-1\}$, are picked from the sorted sample. They are called *splitters*.
- In Phase 2, each processor partitions its keys into P buckets defined by the $P-1$ splitters, and sends, for each $i \in \{1, \ldots, P\}$, the ith bucket to processor i.
- In Phase 3, each processor sorts the keys that it has received during Phase 2.

For all of our implementations, we got the fastest running times for choosing $r = 0.02$. Let $T_{\text{Sample}}(N, P) = T_{\text{Phase 1}} + T_{\text{Phase 2}} + T_{\text{Phase 3}}$. For the predictions, we assume that the local buckets have size $\frac{N}{P^2}$, which is true on average. The two implementations we discuss in the following have identical realizations of Phases 2 and 3. Therefore, we first describe these phases.

After Phase 1, the sorted splitters are stored in a global variable of size P that is accessed at the beginning of Phase 2 by all processors. By a binary search, all keys are put into their respective local buckets. By a parallel prefix, each processor computes the exact positions where it has to place its buckets in the destination processors. Finally, the buckets are written into these positions. Thus,

$$\begin{aligned} T_{\text{Phase 2}} &= 2 \cdot \text{agv}(k^*, P, P) + \text{agv}(k^*, \tfrac{N}{P^2}, P) + 2 \cdot \text{sync}(k^*, P) \\ &\quad + 18.7 \cdot \tfrac{N}{P} \cdot \log P + \text{GlobVis}(P) + 4.1 \cdot P^2 \ . \end{aligned}$$

Phase 3 applies Quicksort to sort the keys per processor sequentially. Thus,

$$T_{\text{Phase 3}} = 2 \cdot \text{GlobVis}(\tfrac{N}{P}) + t_{\text{qsort}}(\tfrac{N}{P}) \ .$$

In the following, we describe two different implementations of the first phase.

Sorting the sample sequentially. In the first implementation (Samplesort I), Phase 1 is realized by sending the whole sample to processor 0. It sorts the sample by Quicksort. This leads to the following prediction:

$$\begin{aligned} T_{\text{Phase 1}} &= 108 \cdot (0.02 \cdot \tfrac{N}{P}) + \text{GlobVis}(0.02 \cdot \tfrac{N}{P}) + \text{agv}(k^*, 0.02 \cdot \tfrac{N}{P}, 1) \\ &\quad + t_{\text{qsort}}(0.02 \cdot N) + 3.5 \cdot P + \text{GlobVis}(P) + 2 \cdot \text{sync}(k^*, P) \end{aligned}$$

Sorting the sample in parallel. In the second implementation (Samplesort II), the sample is sorted by applying Odd-Even Merge Sort with guarded S&M operation (see Section 4.1.2). Blelloch *et al.* [6] have implemented a Samplesort where the sample is sorted by a parallel Radixsort. After this sorting, each processor (except for one) holds one splitter.

$$\begin{aligned} T_{\text{Phase 1}} &= 108 \cdot (0.02 \cdot \tfrac{N}{P}) + \text{GlobVis}(0.02 \cdot \tfrac{N}{P}) \\ &\quad + T_{\text{Rüb}}(P, 0.02 \cdot N) + \text{sync}(k^*, P) + \text{agv}(k^*, 1, 1) + \text{GlobVis}(P) \end{aligned}$$

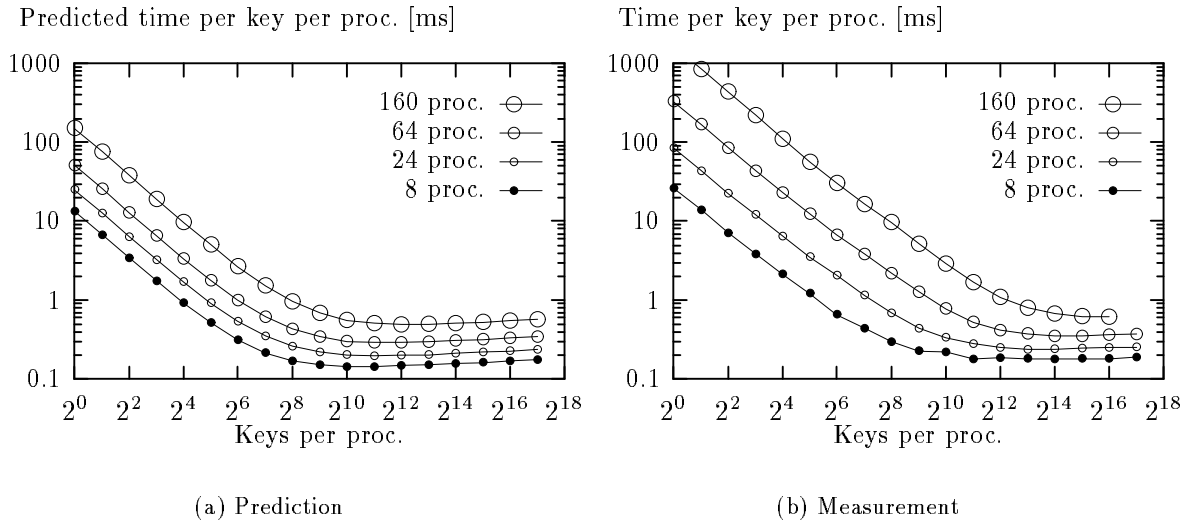
If $T_{\text{Rüb}}(P, 0.02 \cdot N) < t_{\text{qsort}}(0.02 \cdot N)$, this Phase 1 is, for large N/P, significantly faster than sorting the sample sequentially. In fact, it is about 40% faster than the first variant (see Figures 7 and 8). Note that the application of Odd-Even Merge Sort in Phase 1 was realizable with OCCAM-light easily.

A third variant that combines Phases 1 and 2 and that we have investigated reserves a special processor for sorting the sample sequentially. Simultaneously, all other processors already sort their keys so that, during Phase 2, the binary search can be replaced by a single linear scan. But the prediction as well as the implementation show that this variant is only slightly faster than the first variant and considerably slower than the second variant.

Figure 9 shows that the ratio between communication and internal calculation is significantly improved compared to Odd-Even Merge Sort with guarded S&M (Figure 5).

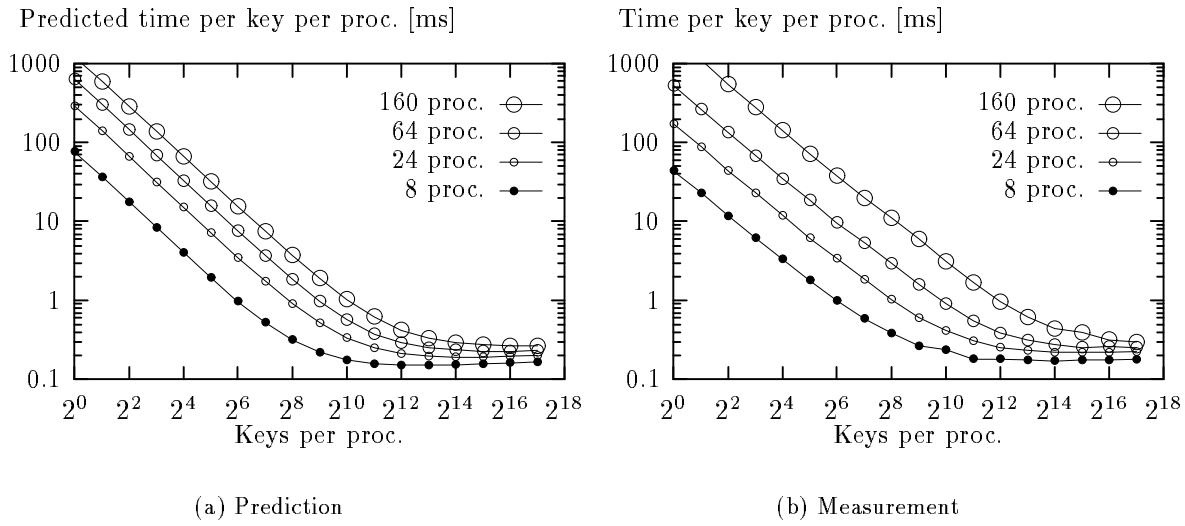
Figure 10 shows when Samplesort II becomes faster than Bitonic Sort on the 5-dimensional butterfly network, based on predictions and measurements. In this figure, we refrained from using Odd-Even Merge Sort with guarded S&M operation, because of the somewhat inexact prediction due to the theoretical analysis.

Obviously, it cannot be guaranteed that all processors hold the same number of keys at the end. If this is necessary, an additional "load balancing" phase has to be performed. It results in a runtime increase of about 10%.



(a) Prediction (b) Measurement

Figure 7: Performance of Samplesort I.



(a) Prediction (b) Measurement

Figure 8: Performance of Samplesort II.

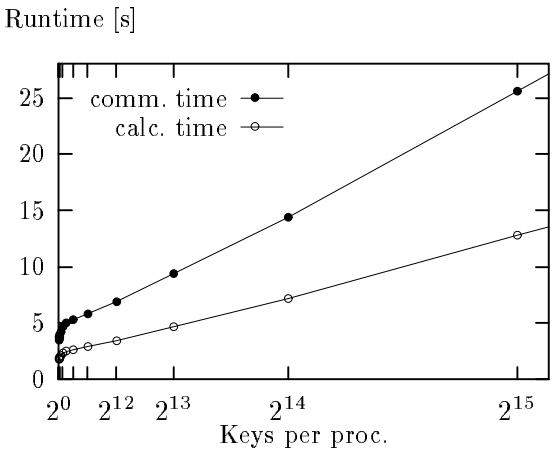


Figure 9: Time spent for calculation and communication in Samplesort II on the 5-dimensional butterfly network (cumulative).

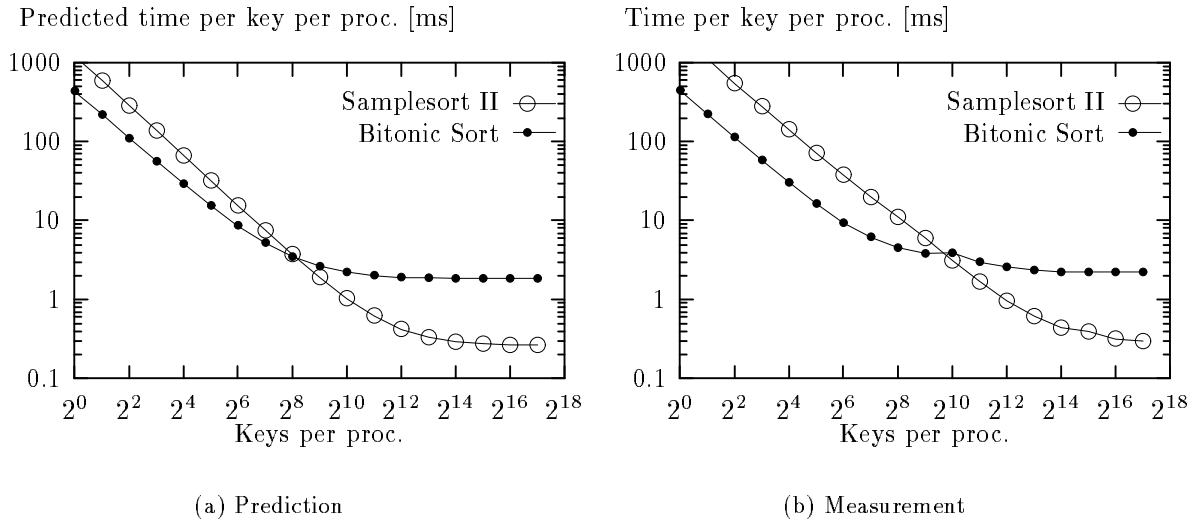


Figure 10: When Samplesort II becomes faster than Bitonic Sort on the 160 processor network.

5 Conclusions

Figure 11 shows which algorithm is the fastest when N/P and P are given. It is shown that, for large sorting problems, Samplesort II is the fastest method, for problems of medium size, Rüb's method is the fastest, and for small problems, Odd-Even Merge Sort should be used.

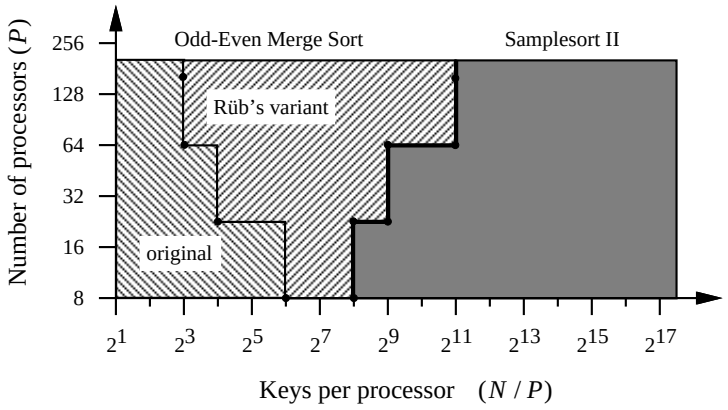


Figure 11: Which algorithm performs best with respect to N/P and P.

The aim of the work of this paper was not to optimize the code for the algorithms, but to demonstrate that it is possible to easily write efficient parallel programs and to rate their behavior by a relatively simple model. We only described simple changes in the algorithms. Once the hardware platform is chosen, the model enables us to give good estimations for the influence of theses changes. The model is sufficient when the algorithms have a relatively large amount of communication (see Figure 5), as well as when the amount of communication is relatively small (see Figure 9).

Acknowledgments

We would like to thank Reinhard Lüling, Friedhelm Meyer auf der Heide, Brigitte Oesterdiekhoff, and Berthold Vöcking for fruitful discussions.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \cdot \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Moffet Field, Mar. 1994.

[3] D. H. Bailey, J. T. Barton, T. Lasinski, and H. D. Simon. NAS parallel benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, Moffet Field, Jan. 1991.

[4] K. E. Batcher. Sorting networks and their applications. In *AFIPS Conf. Proc. 32*, pages 307–314, 1968.

[5] G. E. Blelloch, L. Dagum, S. J. Smith, K. Thearling, and M. Zagha. An evaluation of sorting as a supercomputer benchmark. Technical Report RNR-93-002, NASA Ames Research Center, Moffet Field, Jan. 1993.

[6] 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 3rd ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 3–16, 1991.

[7] 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 (HICSS)*, volume I, pages 200–208, 1997.

[8] R. Cole. Parallel merge sort. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, pages 453–495. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[9] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Commun. ACM*, 39(11):78–85, 1996.

[10] D. E. Culler, A. Dusseau, R. Martin, and K. E. Schauser. Fast parallel sorting under LogP: from theory to practice. In *Portability and Performance for Parallel Processing*, pages 71–98, 1994.

[11] R. Diekmann, J. Gehring, R. Lüling, B. Monien, M. Nübel, and R. Wanka. Sorting large data sets on a massively parallel system. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 2–9, 1994.

[12] M. Dowd, Y. Perl, M. Saks, and L. Rudolph. The periodic balanced sorting network. *J. ACM*, 36:738–757, 1989.

[13] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin. Fast parallel sorting under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems*, 7:791–805, 1996.

[14] M. Fischer, J. Rethmann, and A. Wachsmann. A realistic cost model for the communication time in parallel programs. In *Proceedings of the 3rd Workshop on Abstract Machine Models for Parallel and Distributed Computing (AMW)*, pages 13–27, 1996.

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

[16] R. Funke, R. Lüling, B. Monien, F. Lücking, and H. Blanke-Bohne. An optimized reconfigurable architecture for Transputer networks. In *Proceedings of the 25th Hawaii Int. Conf. on System Sciences (HICSS)*, volume I, pages 237–245, 1992.

[17] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

[18] M. T. Goodrich. Communication-efficient parallel sorting. In *Proceedings of the 28th ACM Symposium on Theory of Computing (STOC)*, pages 247–256, 1996.

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

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

[21] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Trans. Comput.*, 34:344–354, 1985.

[22] The NAS parallel benchmarks home page. `http://science.nas.nasa.gov/Software/NPB/`.

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

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

[25] C. Rüb. On the average running time of odd-even merge sort. *Journal of Algorithms*, 22:329–346, 1997.

[26] D. Sanders, Y. Park, and V. Govindan. Parallel sorting on the NEC Cenju-3 and IBM SP2. In *Proceedings of the International Conference on High-Performance Computing in Asia (HPC Asia)*, pages 214–219, 1997.

[27] T. M. Stricker. Supporting the hypercube programming model on mesh architectures (A fast sorter for iWarp tori). In *Proceedings of 4th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 148–157, 1992.

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

[29] A. Wachsmann. *Eine Bibliothek von Basisdiensten für Parallelrechner: Routing, Synchronisation, gemeinsamer Speicher.* Dissertation, Universität-GH Paderborn, 1995. In German.

[30] A. Wachsmann and F. Wichmann. OCCAM-light — A multiparadigm programming language for Transputer networks. Forschungsbericht der Forschergruppe "Effiziente Nutzung massiv paralleler Systeme" Nr. 5, Universität-GH Paderborn, April 1993. Available by anonymous ftp from `ftp.uni-paderborn.de` as file `/doc/techreports/Informatik/tr-rf-93-005.ps.Z`.