Optimal Average Case Sorting on Arrays*

Manfred Kunde[2], Rolf Niedermeier[1], Klaus Reinhardt[1], and Peter Rossmanith[2]

[1] Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Sand 13, D-72076 Tübingen, Fed. Rep. of Germany
[2] Fakultät für Informatik, Technische Universität München, Arcisstr. 21, D-80290 München, Fed. Rep. of Germany

Abstract. We present algorithms for sorting and routing on two-dimensional mesh-connected parallel architectures that are optimal on average. If one processor has many packets then we asymptotically halve the up to now best running times. For a load of one optimal algorithms are known for the mesh. We improve this to a load of eight without increasing the running time. For tori no optimal algorithms were known even for a load of one. Our algorithm is optimal for every load. Other architectures we consider include meshes with diagonals and reconfigurable meshes. Furthermore, the method applies to meshes of arbitrary higher dimensions and also enables optimal solutions for the routing problem.

1 Introduction

We present deterministic algorithms that sort and route on mesh-connected computers fast on average. For important, fundamental classes of problems (so called h–h relations) we completely solve the problem in that sense that our approach is optimal for all cases. (We present matching lower bounds.)

A two-dimensional mesh-connected computer is a processor array, where each processor has one bidirectional connection to each of its four neighbors. Meshes are a promising parallel architecture due to their scalability, their regular interconnection structure with its locality of communication, and since they need only linear space in the VLSI-model. We also consider meshes with wrap-around connections, also known as tori, meshes with additional diagonal connections, and reconfigurable meshes.

Average case analysis is in general more difficult than worst case analysis. This is the reason why there are much more results about the worst case behavior than about the average case for many parallel models. Often, however, the average case behavior is the more realistic one and an algorithm good in the average is usually superior to an algorithm whose good worst-case behavior is at its average case behavior's cost. A good example is Quicksort [9].

Our algorithms are based on the following simple principle: Let us for the moment assume that the input consists only of zeros and ones. We can assume

* This research was supported by the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 0342, TP A4 "KLARA."

that on average the input is very uniformly distributed over the whole mesh. If we divide the mesh into equal-sized blocks, the relative number of ones in each block will be about the same. Sorting the blocks leads to a situation where all blocks look like nearly the same. There are zeros, followed by ones. The point where the zeros end and the ones begin is about the same in each block.

Let us assume that there are k blocks and let us also divide each block into k *bricks.* Now the first bricks of all blocks will contain only zeros, the second bricks, too. The last bricks of all blocks will contain only ones. Around the, say, ith brick, however, zeros end and ones begin. In some blocks it might be the ith brick, in others the $i+1$st, in others the $i-1$st, but we can assume that in all blocks this happens somewhere near the ith brick. Next, we perform a so-called *all-to-all mapping* that sends the first bricks of all blocks into the first block, the second bricks of all blocks into the second block, and so on. Now the first *block* contains only zeros, the last block only ones. Only around the ith block, ones and zeros occur together. The input is therefore almost sorted.

To finish, we must sort some adjacent blocks and we are done. (Note that in this paper we will not describe the concrete realization of all-to-all mappings. For this purpose we just refer to the literature [10, 11].)

Of course, this does not work for *all* inputs, but it works for *most* inputs. We will see that the probability that it works is exponentially near to one.

When all-to-all mappings were introduced, for the first time an algorithm matching the trivial bisection bound came up [10]. This algorithm solves the h–h sorting problem optimally on meshes, where $h \geq 8$. That is, each processor has at most h data packets in the beginning and the end. The input is sorted according to an *indexing*, that is, an ordering of all places in the grid. The algorithms based on all-to-all mappings work for a lot of different indexing schemes. The bisection bound is based on the fact that in the worst case the upper half and the lower half of the mesh have to be swapped. Then $hn^2/2$ data packets must travel over n communication links, which obviously takes at least $hn/2$ steps.

In the average case the bisection bound does not hold. So we can improve upon algorithms that even match this lower bound. For a long time no algorithm came even close to the bisection bound and up to now algorithms matching the bisection bound asymptotically were state of the art. Now we improve upon these algorithms; actually we even *halve* their running times. For $h < 8$ also no optimal algorithm are known in general, but the 1–1 problem was solved by Chlebus [1] in asymptotically optimal $2n + o(n)$ steps on average. We show that up to $h = 8$ we can sort within the same time bound. Recently, Kaufmann, Sibeyn, and Suel [7] showed that the 1–1 problem can be solved within the same time bound even in the worst case. The good worst case behavior is, however, at the expense of big constants, especially a buffer size of the processors around 25. Chlebus' algorithm and ours are superior; they only need buffer size 1.

In the case of mesh-connected tori the best known algorithms with buffer size 1 solve the 1–1 sorting problem in $1.5n + o(n)$ steps for a blocked snake-like row major order, and in $2n + o(n)$ steps for a row major and a snake-like row major order [3]. Gu and Gu improved herein the previous results each by $0.5n$.

We improve their running times by another $0.5n$ to $n + o(n)$, $1.5n + o(n)$, and $1.5n + o(n)$, respectively. Instead of a blocked snake-like row major we may use other blocked indexing schemes, as well. We thereby also refute a conjecture of Gu and Gu: They conjectured that $n + o(n)$ steps cannot be reached for tori.

Our algorithm and its probabilistic analysis differs significantly from the algorithms and proofs given by Chlebus [1] and Gu and Gu [3].

Fast all-to-all mappings are also known for higher dimensional grids [10], grids with diagonals [11], and reconfigurable meshes [6]. Therefore, we also obtain optimal results for average case sorting on these models of parallel computation.

The following table summarizes the running times for all models considered in this paper. Our algorithms are optimal with respect to asymptotic running times and in most cases also with respect to buffer sizes.

Problem	Model	New Result		Old Result	
1–1 sorting	mesh	$2n$		$2n$	[1]
	torus	n		$1.5n$	[3]
h–h sorting	mesh	$2n$,	$h \leq 8$	$4n$	[10]
		$hn/4$,	$h > 8$	$hn/2$	[10]
	torus	n,	$h \leq 8$	$2n$	[10]
		$hn/8$,	$h > 8$	$hn/4$	[10]
	mesh with diagonals	n,	$h \leq 9$	$2n$	[11]
		n,	$h \leq 12$	$hn \cdot 2/9$	[11]
		$hn/12$,	$h > 12$	$hn \cdot 2/9$	[11]
	torus with diagonals	$n/2$,	$h \leq 10$	n	[11]
		$n/2$,	$h \leq 12$	$hn/9$	[11]
		$hn/24$,	$h > 12$	$hn/9$	[11]
	reconfigurable mesh	$hn/2$,	$h \geq 1$	hn	[6]

Additive low order terms in the running times are omitted.
For $h \geq 2$ no average case running times faster than for the worst-case were known.

In the next section we provide basic notation and some more facts about our models of computation. In the third section we present our algorithm for average case sorting. In the fourth section we analyze the average case behavior of our algorithm on independent random inputs from the unit interval and on random permutations of sets and multisets, and finally prove matching lower bounds. In the fifth section we explain the applications of our findings to tori, grids with diagonals, and reconfigurable meshes and also to the routing problem. We end with some conclusions and open questions in the sixth section.

2 Preliminaries

Leighton [12, 13] provides a general view of most of the topics we deal with. We assume familiarity with basic facts and concepts of elementary probability theory [2, 5].

A processor grid (or, equivalently, mesh) consists of n^2 processors arranged in a two-dimensional array. All processors work in parallel synchronously, and only nearest neighbors may exchange data. For h–h problems each processor contains at most h data elements in the beginning and at the end of the algorithm. Each element in a processor lies in one of h local places. A processor is said to have buffer size s if it can retain at most s elements during the whole computation. For h–h sorting problems we need an indexing scheme for the processor places. An index function numbers the places of the grid processors from 0 to $hn^2 - 1$. The ith smallest element will be transported to the place indexed $i-1$. In one step a communication channel can transport at most one element (as an atomic unit) in each direction. For complexity considerations we only count communication steps; we ignore operations within a processor.

From the description of a grid we easily obtain several further parallel architectures to be considered in this paper. So we get processor tori simply by adding wrap-around connections to processor grids. Grids or tori with diagonals evolve if we add diagonal connections, replacing four-neighborhoods by eight-neighborhoods. Eventually, reconfigurable grids differ from conventional grids in that they have switches with reconnection capability. So they build a grid-shaped reconfigurable bus system, which is changeable by adjusting the local connections between ports within each processor. A communication through a bus is assumed to take unit time no matter how long the bus is. For details we refer to e.g. [6, 14].

In order to describe our algorithm, we introduce some more concepts concerning the partitioning of grids into subgrids and indexing schemes. We subdivide an $n \times n$ grid of processors into $k = n^{2/3-2\epsilon}$ subgrids of $b = n^{2/3+\epsilon} \times n^{2/3+\epsilon}$ processors, called *blocks*. Here ϵ is some real constant with $0 < \epsilon < 1/3$. Each block is subdivided into k *bricks* of $n^{1/3+2\epsilon} \times n^{1/3+2\epsilon}$ processors.

The algorithm will sort according to any indexing fulfilling the following requirements. If we number the blocks from 1 to k, then blocks i and $i+1$ have to be neighbors and the indexing is chosen in such a way that all places in block i have smaller indices than places in block $i+1$. That means the indexing is *continuous* with respect to blocks.

All-to-all mappings [10], as described in the introduction, are employed to exchange bricks between all pairs of blocks in the grid. As it will turn out in the next section, the time complexity of the all-to-all mapping dominates the overall complexity of our average case sorting algorithm. So, e.g., the existence of algorithms realizing all-to-all mappings for h–h relations with $h \leq 8$ on meshes in $2n$ steps [10] implies that our algorithm in this case needs time $2n + o(n)$.

3 The Algorithm

Our Algorithm A for average case sorting is based on the principle of sorting with *all-to-all mappings* [10, 11]. Roughly speaking, an all-to-all mapping performs a global distribution of data by sending a brick from each block to each other block of the grid. A quick and raw description of sorting with all-to-all mappings then looks as follows. First sort each block individually, then perform an all-to-all mapping, then again sort each block individually, then again perform an all-to-all mapping and finally sort all adjacent pairs of blocks (Algorithm B). Let us only mention in pass that the time complexity of that algorithm is clearly predominated by the complexity of the two all-to-all mappings. We omit any further details concerning the above algorithm and now immediately start with our algorithm for average case sorting.

In simplified terms, Algorithm A works as follows. It consists of mainly two parts. First, we use a cut version of Algorithm B with only *one* all-to-all mapping. Second, we apply a worst case linear time sorting algorithm, which, however, is only executed if the first part fails to sort the input. In the following section we will show that failure of the first part is extremely unlikely.

Before we sketch the details of our algorithm, note that in each individual sorting of a block we sort according to an indexing that again is continuous. Let us number the bricks of a block from 1 to k. Then the block local indexing is chosen in such a way that all places in brick i have smaller indices than all places in brick $i+1$. The ith brick of each block will be sent to the ith block of the grid by means of an all-to-all mapping. Algorithm A works for both grids and tori.

Algorithm A.
1.1. Sort each block.
1.2. All-to-all mapping.
1.3. Sort all adjacent pairs of blocks.
2. If elements are out of order, sort again with some other linear time algorithm.

Algorithm B. [10]
1. Sort each block.
2. All-to-all mapping.
3. Sort each block.
4. All-to-all mapping.
5. Sort adjacent pairs of blocks.

Now we analyze the time complexity of Algorithm A. Steps 1.1 and 1.3 take time proportional to the side length of a block, i.e., $O(n^{2/3+\epsilon})$. Step 1.2 takes time $2n$ for $h \leq 8$ and it takes time $hn/4$ for $h > 8$. Step 2 takes linear time. Since we will show in the next section that it is extremely unlikely for randomly given inputs that step 2 will be executed, we can derive an average case running time of $2n + O(n^{2/3+\epsilon})$ for $h \leq 8$ and $hn/4 + O(n^{2/3+\epsilon})$ for $h > 8$ in the case of grids.

Let us end this section with a case of special interest: 1–1 sorting with buffer size 1. It is straightforward to see that for small, constant h our algorithm always works with small, constant buffer size. For 1–1 problems we even may perform the all-to-all mapping with buffer size 1, which implies an average case sorting algorithm on grids with buffer size 1 running in time $2n + O(n^{2/3+\epsilon})$. For the case of tori, where we get running time $n+O(n^{2/3+\epsilon})$ this is a direct consequence of the fact that the shift operations used in the original algorithm [10] can be implemented using cyclic shifts in rows and columns. We omit the details.

For the case of grids, the situation is a bit more delicate. The basic idea here is as follows. Partition the grid horizontally into two equally sized parts. We transport half of the elements of the lower part to the upper part and vice versa as follows. Subdivide the grids vertically into strips of width 2 and length n. Within the strips perform a rotary traffic of the elements in the natural way. It takes time $n/2$ to get half of the elements of the lower strip to the upper and vice versa. Next we perform the same idea for the exchange of half of the elements between the upper two quarters and between the lower two quarters of the grid each time. This again takes $n/2$ steps. Eventually we recursively perform the same method within each of the four quarters of the grid and so on. We defer the details to the full paper.

4 Analysis

In this section we provide the probabilistic analysis of Algorithm A. We proceed as follows. First, we show that the first part of Algorithm A correctly sorts independently chosen random numbers between 0 and 1 with overwhelming probability. Next, we prove that this also implies the same behavior when the input is a uniformly chosen permutation of a set or multiset of hn^2 fixed elements. Finally, we state our matching lower bounds for average case sorting on grids.

4.1 Sorting random numbers

Definition 1. An algorithm T-separates an input $x_1, \ldots, x_N$ if it permutes it into $y_1, \ldots, y_N$ such that $y_i < T$ and $y_j \geq T$ implies $i < j$.

To T-separate a set of data is a step forward to sorting the set. For example, Quicksort sorts by recursively T-separating for certain T's. Obviously, if $y_1, \ldots, y_N$ are simultaneously y_1-, y_2-,..., and y_N-separated, then they are sorted, since no pair is out of order. In the following, we show that the first part of Algorithm A sorts with high probability by proving that it T-separates for all T's with high probability.

Lemma 2. *If $X_1, \ldots, X_N$ are independent random variables that are uniformly distributed over the unit interval, then the first part of Algorithm A with block size b and number of blocks k will T-separate the input $X_1, \ldots, X_N$ with probability at least $1 - 2ke^{-b/(4k^2)}$ for a fixed $T \in [0, 1]$.*

Proof. We focus on the first block containing elements $X_1, \ldots, X_b$. After sorting the blocks, these elements are in order. Let us call a brick *dirty* if it contains both elements smaller than T and greater than or equal to T. There is at most one dirty brick because the indexing of places follows the numbering of bricks. Let Z be the number of elements in the block that are smaller than T. Since $Pr(X_i \leq T) = T$, the expected value of Z is Tb. We apply Chernoff-bounds in order to estimate the probability that Z deviates more than a half brick's size $b/(2k)$ from its expectation. We use inequalities (1) and (12) of Hagerup and Rüb [4]. Let $S := 1 - T$.

$$
\begin{aligned}
&Pr(\, Z \geq (T + 1/(2k))\, b\,) \leq \\
&\leq \left(\frac{Tb}{(T+1/(2k))b}\right)^{(T+1/(2k))b} \left(\frac{Sb}{(S-1/(2k))b}\right)^{(S-1/(2k))b} \\
&= \left(1 + \frac{1}{(2kT)}\right)^{-(T+1/(2k))b} \left(1 - \frac{1}{(2kS)}\right)^{-(S-1/(2k))b} \\
&\leq \exp(-\tfrac{1}{2kT}(T + 1/(2k))b) \cdot \exp(\tfrac{1}{2kS}(S - 1/(2k))b) \\
&\leq e^{-b/(4k^2)}
\end{aligned}
$$

Similarly,

$$Pr(\, Z \leq (T - 1/(2k))\, b\,) \leq e^{-b/(4k^2)}$$

follows. Together we have

$$Pr(\, |Z - Tb| \geq b/(2k)\,) \leq 2e^{-b/(4k^2)}. \qquad (*)$$

If $Tb - b/(2k) < Z < Tb + b/(2k)$, then the dirty brick is the ith brick, where $i = \lceil \frac{Z}{b/k} \rceil$, i.e.,

$$i \in \{\lceil Tk - \tfrac{1}{2} \rceil, \lceil Tk + \tfrac{1}{2} \rceil\}. \qquad (**)$$

The exact position does not matter; it is only important that we can spot the dirty brick in a small region with high probability. For all other blocks, the probability that the dirty brick lies in one of the two positions is the same. Therefore, the probability that the dirty brick in *all* k blocks is in one of the two positions is at least $1 - 2ke^{-b/(4k^2)}$. If this is the case, then the positions of the *dirty blocks* after the all-to-all mapping are also restricted to $(**)$. Local sorting of adjacent blocks at the end of Algorithm A then "cleans" these dirty blocks. □

It remains to generalize Lemma 2 from some fixed T to all values of interest.

Lemma 3. *If $X_1, \ldots, X_N$ are independent random variables that are uniformly distributed over the unit interval, then the first part of Algorithm A will T-separate the input $X_1, \ldots, X_N$ with probability at least $1 - 2Nke^{-b/(4k^2)}$ simultaneously for all $T \in [0,1]$.*

Proof. If the algorithm X_1-, X_2-, …, and X_N-separates the input, then it T-separates it for *all* $T \in [0,1]$. So by Lemma 2 the probability is at least $1 - 2Nke^{-b/(4k^2)}$. □

We summarize our findings in the following corollary, employing $N = hn^2$ and the given values for b and k.

Corollary 4. *The first part of Algorithm A sorts random inputs from the unit interval with probability at least $1 - 2hn^{8/3-2\epsilon}e^{-n^{1.5\epsilon}}$.*

4.2 Sorting permutations of sets and multisets

In the previous subsection we proved the correctness and efficiency of our algorithm on input elements that were given by independently chosen random numbers from the infinite probability space $[0,1]^{hn^2}$. As a direct consequence our algorithm is also suitable to sort hn^2 fixed elements that are in random order. We start with the case of hn^2 distinct elements and give a rigorous proof, though the consequence might seem obvious.

Theorem 5. *In the average, Algorithm A sorts a permutation of a set of hn^2 distinct elements on grids in $2n + O(n^{2/3+\epsilon})$ steps if $h \leq 8$ and in $hn/4 + O(n^{2/3+\epsilon})$ steps if $h > 8$.*

Proof. Let $N = hn^2$ and $x_1, \ldots, x_N \in [0,1]$. Let V be the set of all vectors $(x_1, \ldots, x_N) \in [0,1]^N$, where all x_i are pairwise distinct. The probability that a vector chosen randomly from $[0,1]^N$ lies in V is 1. The set V can now be partitioned into $N!$ disjoint sets by the surjective mapping $P : V \to S_N$, where S_N is the set of permutations of $\{1, \ldots, N\}$ and $P(x_1, \ldots, x_N) = \pi$ iff $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(N)}$. That is, $P^{-1}(\pi)$ is the set of those inputs $x_1, \ldots, x_N$ that are sorted by π. Since Algorithm A is comparison-based, we get immediately: If $P(x_1, \ldots, x_N) = P(y_1, \ldots, y_N)$ then the first part of Algorithm A sorts $x_1, \ldots, x_N$ correctly if and only if it sorts $y_1, \ldots, y_N$ correctly.

Now let $\rho \in S_N$ be an arbitrary permutation. The probability that a randomly chosen vector $x_1, \ldots, x_N$ from V lies in $P^{-1}(\rho)$ is $1/N!$ [8, page 64], i.e., $Pr((x_1, \ldots, x_N) \in P^{-1}(\rho)) = 1/N!$. Let I denote the set of "good" input vectors, i.e., the set of vectors from $[0,1]^N$ that are sorted correctly by the first part of Algorithm A and let G be the set of "good" permutations:

$$G := \{\, P(x_1, \ldots, x_N) \mid (x_1, \ldots, x_N) \in I \cap V \,\}$$

Then

$$\begin{aligned} Pr(\mathbf{x} \in I \cap V) &= Pr(\mathbf{x} \in \bigcup_{\pi \in G} P^{-1}(\pi)) \\ &= \sum_{\pi \in G} Pr(\mathbf{x} \in P^{-1}(\pi)) = |G|/N! \end{aligned}$$

The probability that a uniformly chosen permutation is a good permutation is exactly $|G|/N!$. Since $Pr(\mathbf{x} \notin V) = 0$ and by Lemma 3 we conclude that $|G|/N! = Pr(\mathbf{x} \in I \cap V) = Pr(\mathbf{x} \in I) \geq 1 - 2Nke^{-b/(4k^2)}$. □

We continue with the case of multiset sorting, putting it down to the case of permutations on sets.

Theorem 6. *In the average, Algorithm A sorts a permutation of a multiset of hn^2 elements on grids in $2n + O(n^{2/3+\epsilon})$ steps if $h \leq 8$ and in $hn/4 + O(n^{2/3+\epsilon})$ steps if $h > 8$.*

Proof. Assume that the input consists of hn^2 not necessarily distinct elements. Replace each element x by a pair (x, z), where the first component is the original element and the second component is a random number z uniformly chosen from $[0, 1]$. Let us further define $(x_1, z_1) < (x_2, z_2)$ iff $x_1 < x_2$ or $x_1 = x_2$ and $z_1 < z_2$. Then all pairs are distinct from each other with probability 1 and we may follow exactly the same line of argumentation as in Theorem 5 to show that our algorithm sorts the pair elements according to the newly defined ordering $<$. Herein observe that since all permutations over the multiset are equally likely, the order of the pairs according to $<$ is also random.

It remains to be shown that sorting the pairs implies sorting the original input. It suffices to observe that pair elements with same first components will stand side by side after the above sorting and removing the second components of all pairs yields the multiset in sorted order. On the other hand, our choice of $<$ does at most affect the order of subsequences of equal elements. □

4.3 Lower bounds

Chlebus [1] showed that the distance bound for grids is a lower bound also in the average case, by proving that with high probability an element near the upper left corner must travel near to the lower right corner. The lower bound holds also for h–h problems with $h > 1$. For $h \leq 8$ we have algorithms that sort in $2n + O(n^{3/4})$ steps; these algorithms are therefore asymptotically optimal.

For $h > 8$ our algorithms need asymptotically more than $2n$ steps. Still we show that even for $h > 8$ the algorithms remain optimal by proving a new lower bound that asymptotically matches the running time of our best algorithms.

Theorem 7. *Every algorithm that solves the h–h sorting problem on a grid for all $\delta > 0$ needs at least $hn/4 - O(n^{1/3+\delta})$ steps on average.*

Proof. Divide the grid horizontally into two halves and choose an indexing such that no block is divided and that only two neighboring blocks are divided. Figure 1 shows a blockwise snake-like ordering as an example. The first part of Algorithm A sorts correctly with very high probability and moves at least $hn^2/4 - O(n^{4/3+2\epsilon})$ packets from the upper half to the lower half, because an all-to-all mapping moves $hn^2/4$ packets in the respective other half and a local sort moves up to one block back.

Look again at the figure. Since only adjacent pairs of blocks are sorted after the all-to-all mapping took place, only the contents of block 33 might travel back into the upper half. That means that in total half of the contents of 32 blocks travelled from the upper into the lower half and then at most one block back into the upper half.

With very small probability the first part of Algorithm A does not sort correctly and the second part of Algorithm A moves up to $hn^2/2$ packets from the upper half to the lower half. Anyways, the expected value of packets that travel from the upper into the lower half remains $hn^2/4 - O(n^{4/3+2\epsilon})$. This amount of packets has to be transported through the dividing line that consists of only n

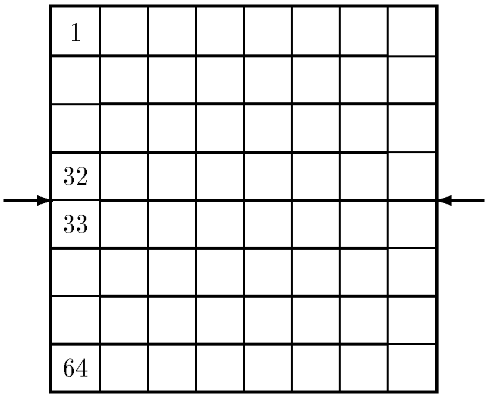

Fig. 1. The lower bounds argument

wires. So only n packets travel down in one step and thus $hn/4 - O(n^{1/3+2\epsilon})$ steps are necessary. This holds for *every* sorting algorithm, since every algorithm that sorts must perform the same transports if all input elements are distinct (though perhaps in a different sequence). □

Note that in the above proof we made indirect use of the upper bound provided by Algorithm A in order to prove the lower bound.

5 Applications

A closer look at our algorithm and its analysis reveals that the fundamental properties we used were the partitioning of the grid into blocks and bricks and the existence of an efficient all-to-all mapping. Thus we may apply our algorithmic idea together with its analysis to all cases where an efficient all-to-all mapping exists. With other words: We can replace Algorithm B by Algorithm A independently of the underlying parallel computer. In this way we get results for higher dimensional meshes and tori [10], meshes with additional diagonal connections [11], and reconfigurable meshes [6]. The table in the introduction provides a summary of all our results discussed in the following.

Tori. For tori (meshes with wraparound connections) we simply make use of the known fact that the all-to-all mapping can be implemented twice as fast as for conventional grids [10]. That yields a complexity of $n + O(n^{2/3+\epsilon})$ for h–h problems if $h \leq 8$ and $hn/8 + O(n^{2/3+\epsilon})$ if $h > 8$. These results are optimal for all h and improve results of Gu and Gu [3] in three aspects. First, we are in general faster than they are by an additive term of $0.5n$ for 1–1 sorting with buffer size 1, second, we deal optimally with h–h problems for general h, and, third, we allow for more general indexing schemes as they do. More precisely, for the three cases of indexing functions for 1–1 sorting with buffer size 1 considered by Gu and Gu—blocked snakelike row major, row major, and snakelike row major order—we have complexities of $n+O(n^{2/3+\epsilon})$, $1.5n+O(n^{2/3+\epsilon})$, and $1.5n+O(n^{2/3+\epsilon})$, respectively. Note that the results for row major and snakelike row major indexing functions can be obtained from the results for blocked snakelike row major indexing in the same way as in [3]. In particular, we disprove the conjecture of Gu and Gu that the lower bound n can be improved by showing that it is tight.

Meshes with diagonals [11]. We obtain for the h–h problem in this model $n + O(n^{2/3+\epsilon})$ if $h \leq 12$ and $hn/12 + O(n^{2/3+\epsilon})$ if $h > 12$. Here we use recent improvements of h–h sorting in $2n$ steps from $h = 9$ (as presented in [11]) to $h = 12$ [unpublished manuscript]. In the case of tori with diagonals we again halve these time bounds. Once more these results for average case sorting on meshes with diagonals are optimal due to the corresponding lower bound derived in complete analogy to the case for conventional grids.

Reconfigurable meshes [6]. The bisection bound here is hn for $h \geq 1$ due to the model assumption that processor links only may be used unidirectionally at a time. For average case sorting we can show that $hn/2 - O(n^{2/3})$ is a lower bound. We match this lower bound up to the additive term $O(n^{2/3+\epsilon})$ noting that an all-to-all mapping here needs $hn/2$ steps. Let us only mention in pass that for $h > 4$ reconfigurable meshes show to be inferior to conventional meshes with respect to average case sorting.

Routing. Our results for sorting also imply the same optimal results for the corresponding routing problems simply by making use of routing by sorting.

Since all upper bounds in our paper have matching lower bounds, improvements will only be possible with respect to the low order term.

6 Conclusion

This paper solves the problem of average case h–h sorting on meshes and tori for all $h \geq 1$ in the sense that we give asymptotically matching upper and lower bounds.

A few questions remain open. One is whether it is possible to improve our algorithm for 1–1 sorting on tori for row major or snakelike row major indexing. We expect that it is possible to give a lower bound of $1.5n$, showing the optimality of our results also for these special cases of indexing functions. Another point is to study input distributions different from the uniform distribution as utilized here. For practical purposes it would be desirable to improve the additional low order terms.

References

1. B. S. Chlebus. Sorting within distance bound on a mesh-connected processor array. In H. Djidjev, editor, *Proc. of International Symposium on Optimal Algorithms*, number 401 in Lecture Notes in Computer Science, pages 232–238, Varna, Bulgaria, May/June 1989. Springer.
2. W. Feller. *An Introduction to Probability Theory and its Applications*, volume I. Wiley, 3d edition, 1968.
3. Q. P. Gu and J. Gu. Algorithms and average time bounds of sorting on a mesh-connected computer. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):308–315, March 1994.
4. T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33:305–308, 1990.
5. M. Hofri. *Probabilistic analysis of algorithms.* Texts and Monographs in Computer Science. Springer-Verlag, 1987.
6. M. Kaufmann, H. Schröder, and J. F. Sibeyn. Routing and sorting on reconfigurable meshes. 1994. To appear in *Parallel Processing Letters.*
7. M. Kaufmann, J. F. Sibeyn, and T. Suel. Derandomizing algorithms for routing and sorting on meshes. In *Proceedings of the 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 669–679, 1994.
8. D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming.* Addison-Wesley, 2nd edition, 1969.
9. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, 1973.
10. M. Kunde. Block gossiping on grids and tori: Sorting and routing match the bisection bound deterministically. In T. Lengauer, editor, *Proceedings of the 1st European Symposium on Algorithms*, number 726 in Lecture Notes in Computer Science, pages 272–283, Bad Honnef, Federal Republic of Germany, September 1993. Springer.
11. M. Kunde, R. Niedermeier, and P. Rossmanith. Faster sorting and routing on grids with diagonals. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *Proceedings of the 11th Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science, pages 225–236. Springer, 1994.
12. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1992.
13. T. Leighton. Methods for message routing in parallel machines. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, pages 77–96, 1992.
14. R. Miller, V. K. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout. Parallel computation on reconfigurable meshes. *IEEE Transactions on Computers*, 42(6):678–692, June 1993.

This article was processed using the LaTeX macro package with LLNCS style