On the Efficient Implementation of Massively Parallel Quicksort

Peter Sanders, Thomas Hansch
Department of Computer Science
University of Karlsruhe, 76128 Karlsruhe, Germany
E-mail: sanders@ira.uka.de

Abstract

Parallel Quicksort is one of the most promising parallel sorting algorithms from the point of view of scalability. However, actual implementations have been limited to very basic versions of the algorithm so far and suffer from a number of deficiencies. We have implemented a high performance variant of parallel Quicksort which incorporates the following optimizations: Stop the recursion at the right time, sort locally first, use accurate yet efficient pivot selection strategies, streamline communication patterns, use locality preserving processor indexing schemes and work with multiple pivots at once. In particular, on mesh-connected computers, the resulting algorithm turns out to be among the best practical sorting methods. It is about three times faster than the basic algorithm and achieves a speedup of 810 on a 1024 processor Parsytec GCel for the NAS parallel sorting benchmark of size 2^{24}. The optimized algorithm can also be shown to be asymptotically optimal on meshes.

1 Introduction

Sorting has always been an important area of research both in theoretical and practical computer science. A disappointing observation is that some of the best practical algorithms like sample-sort are not very good from a theoretical point of view. In particular, these algorithms only work well for large amounts of data, i.e., they do not have a good scalability. On the other hand, there are algorithms which are theoretically very scalable. For example, parallel quicksort [18]. However, implementations of parallel quicksort (e.g. [8, 1]) are so far more a proof of concept than a competitive practical sorting algorithm. For large input sizes, simple implementations suffer from problems associated with load balancing and they require a higher communication bandwidth than other algorithms.

In this paper we describe our experiences with implementing a high performance parallel quicksort. Our work has been influenced by a number of interesting comparisons between parallel sorting algorithms regarding their practical usefulness ([3, 9, 6]). However, rather than screening a number of quite different algorithms, we intensively study on one basic strategy – namely parallel quicksort.

In Section 2 we introduce some notation and describe our experimental setup. The basic algorithm for parallel quicksort is described in Section 3. Section 4 discusses the simple yet important measure to switch from quicksort to some specialized algorithm when few PEs are involved. Section 5 explains why it is advantageous to sort the locally present data before starting the main parallel algorithm. Particularly on mesh-connected machines, the locality preserving PE indexing schemes discussed in Section 6 significantly reduce communication expense. Strategies for selecting good pivots as described in Section 7 turn out to be a prerequisite for simplifications in the data exchange patterns introduced in Section 8. Finally, using multiple pivots at once, opens up a new world of algorithmic variants in Section 9. The bottom line performance of the implementation is compared to previous work in Section 10. After a short discussions of algorithmic variants for more general networks in Section 11, Section 12 discusses the results and outlines additional optimization opportunities. The Appendix contains a selection of the data sheets which give more details of our measurements.

2 Basic terminology

We consider a distributed memory MIMD-machine with P PEs numbered 0 through $P-1$. Often, we additionally assume the interconnection network to be an r-dimensional mesh and that $P^{1/r}$ is a power of 2. (Generalizations are outlined in Section 11.) Initially, each PE holds n data items. After sorting, the PEs should collectively hold a permutation of the data such that the items are sorted locally and items on PE i must not be larger than items on PE j for $i < j$.

All measurements have been done on a Parsytec GCel with 16×16 respectively 32×32 PEs.[1] The programs are written in C and use a thin portable communication layer [17] so that they run on many abstract machines (e.g. MPI, PVM, Parix). On the GCel the parallel operating system Cosy [4] ist used. The data items are generated according to the rules of the NAS Parallel Benchmark [2]. We use the ANSI-C library function `qsort` for sequential sorting.

[1] We would like to thank the Paderborn Center for Parallel Computing for making this machine available.

3 Parallel quicksort

We now describe a portable algorithm for parallel quicksort which is similar to the algorithm described in [11] for 2D-Meshes. The sorting is performed by a recursive procedure which is called on a segment of P' PEs numbered P_0 through $P_0 + P' - 1$. Each PE holds local items d_i for $0 \leq i < n'$. Initially, $P_0 = 0$, $P' = P$ and $n' = n$. However, in general n' may take different values on each PE:

- If $P' = 1$ then sort locally and return.
- Else,
  - Collectively select one pivot p for all P' PEs in the segment.
  - Partition the d_i into n'_s *small* items s_j $(0 \leq j < n'_s)$ and n'_l *large* items l_k $(0 \leq k < n'_l)$ such that $s_j \leq p$ and $l_k \geq p$.
  - Let N'_s denote the total number of small items and N'_l denote the total number of large items. Split the PEs in the segment into

    $$P_{\text{split}} := \text{round}\left(P' \frac{N'_s}{N'_s + N'_l}\right)$$

    PEs for the small items and $P' - P_{\text{split}}$ PEs for the large items.
  - Redistribute the items such that "small" PEs receive only small items and "large" PEs receive only large items. Furthermore, each "small" ("large") PE ends up with $\left\lfloor \frac{N_s}{P_{\text{split}}} \right\rfloor$ or $\left\lceil \frac{N_s}{P_{\text{split}}} \right\rceil$ ($\left\lfloor \frac{N_l}{P' - P_{\text{split}}} \right\rfloor$ or $\left\lceil \frac{N_l}{P' - P_{\text{split}}} \right\rceil$) items.
  - Call quicksort recursively (in parallel) for the segments of "small" and "large" PEs.

The necessary coordination between PEs can be performed using the collective operations broadcast, reduce and prefix sum on segments of PEs. It can be shown that even for a very simple pivot selection strategies (i.e. random pivots), the expected parallel execution time is bounded by

$$\mathbf{E}T_{\text{par}} \in \mathrm{O}(n \log n + \log P(T_{\text{routing}}(n) + T_{\text{coll}}))$$

where $T_{\text{routing}}(n)$ is the time required for exchanging n elments on each PE and T_{coll} is the time required for collective operations. The $n \log n$ term is due to sequential sorting and the $\log P$ factor comes from the expeced recursion depth. For example, on a Butterfly-network, this reduces to $\mathbf{E}T_{\text{par}} \in \mathrm{O}(n(\log n + (\log P)^2))$.

4 Breaking the recursion

When the PEs segments are split, there will in general be a roundoff error with the effect that one segment will get around $n/2$ more data items than the other segment. This error happens for every split and finally the PE which ends up with the largest number of items, dominates the time required for local sorting. We have observed a maximum final load between $1.77n$ and $3.16n$ depending on n and the quality the pivot selection. (For example, refer to data sheets 1 and 12.) This problem can be alleviated by stopping the recursion for segments of small size and switching to a different sorting algorithm. We use merge-splitting sort (e.g. [19]). Depending on other optimizations it turns out to be best to stop the recursion for segment sizes between two and four. The final algorithm has almost no data imbalance for large n.

5 Sort locally first

Imbalance of the final data distribution has several disadvantages, but for large n its main impact on execution time is the increased time for local sorting which grows superlinearly with the imbalance. This problem can be solved by doing the local sorting initially before any communication takes place. To maintain the invariant that all items are sorted locally, the sorted sequences of items received during data redistributions are merged. The additional expense for merging is offset by the time saved for partitioning data. Partitioning is now possible in logarithmic time using binary search. Furthermore, more accurate pivot selection strategies are now feasible because the median of the locally present items can be computed in constant time. For large n (2^{15}), the overall improvement due to this measure is about 20 %. (For example refer to data sheets 37 and 55).

6 Locality preserving indexing schemes

In this section we assume that the interconnection network is a square mesh and $\log\sqrt{P} \in \mathbb{N}$. The usual way to number the PEs of a mesh is the row-major ordering shown in Figure 1-(a), i.e., the PE at row i and column j is numbered $i\sqrt{P} + j$. This has the disadvantage that the diameter of the subnetworks involved in data exchange decreases only very slowly, although the number of PEs involved decreases exponentially with the recursion depth. Even when there are only a constant number of PEs left, they may be at a distance of $\sqrt{P}$.

A slight improvement is the snake-like ordering depicted in Figure 1-(b) where a segment of k consecutive PEs never has a diameter of more than $k-1$. But nevertheless there is only a constant factor decrease in the diameter of segments until the segment sizes fall below $\sqrt{P}$.

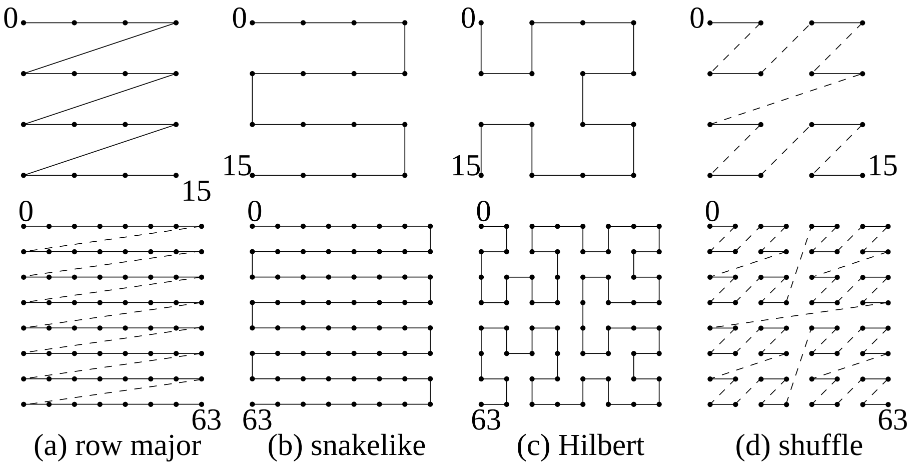


Figure 1: PE numbering schemes for $P = 16$ and $P = 64$.

What we would like to have is a scheme where any segment of PEs numbered $i, \ldots, j$ has a diameter in $O\left(\sqrt{j-i}\right)$. In [5, 14] it is shown that the *Hilbert-indexing* [10] depicted in Figure 1-(c) has a worst case segment diameter of exactly $3\sqrt{j-i} - 2$. (Recently, even better indexing schemes have been found. In [13] it is shown that a scheme based on a Sierpiński-curve has a worst case segment diameter of $\sqrt{8}\sqrt{j-i} - 2$.) In [13] this result is also generalized for a three-dimensional Hilbert-indexing with a diameter of at most $4.73485\sqrt[3]{j-i} - 3$.

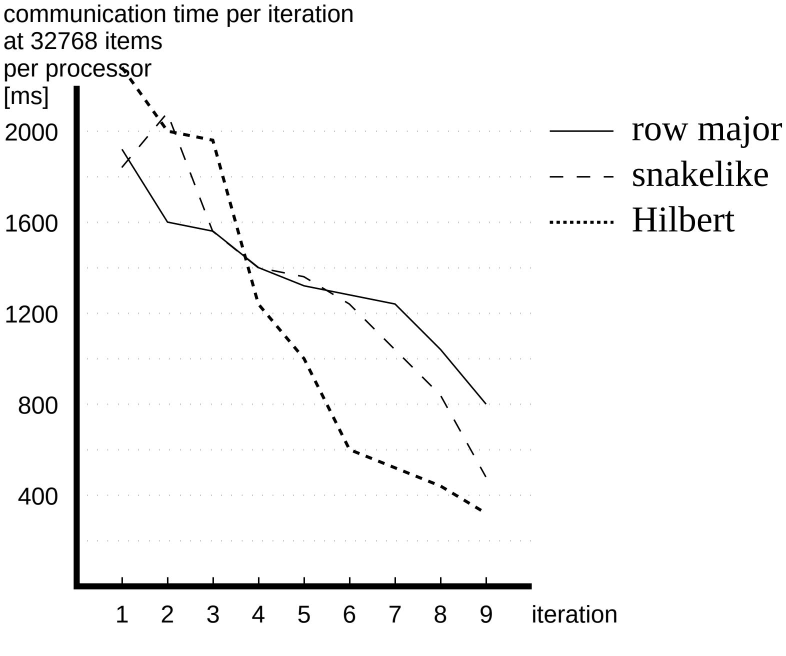


Figure 2: Communication expense for each level of recursion for row-major, snakelike and Hilbert indexing schemes (refer to data sheet 1–3 for details).

The consequence of using these three different indexing schemes is shown in Figure 2. For row-major and snakelike ordering, the communication expense for redistribution decreases only slowly with decreasing segments sizes. For the Hilbert-indexing we have an exponential decrease of communication expense with the recursion depth. This property of the Hilbert indexing also has theoretical appeal. For example, it can be shown that using random pivot selection the execution time is in $\mathrm{O}\left(n \log n + n\sqrt{P}\right)$ with high probability for every legal input. This is asymptotically optimal. Analogous results can be obtained for higher dimensions. However, we observe the practical problem that in the first partitioning step, the Hilbert indexing is slowest. This is due to a more irregular communication pattern and offsets the advantage of the Hilbert indexing for small P.

In conjunction with the optimizations described in the next two sections, the shuffle indexing scheme depicted in Figure 1-(d) (also refer to [11]) is also a good choice. Although it is not locality preserving in the strict sense used above, its self similar and regular structure implies some advantages for algorithms exploiting this structure.

7 Pivot selection

We have implemented a number of increasingly accurate pivot selection strategies. In the simplest case we take the *3-median* of d_0, $d_{n'/2}$ and $d_{n'}$ on PE P_0. On the other end of the spectrum, we use the median of the local medians of all PEs. For small n (e.g. $n = 128$) local strategies are faster. Nevertheless, it always pays off to invest more effort in median selection than using the 3-median. For large n, the more accurate global strategies are always preferable. Very accurate pivot selection strategies are also a prerequisite for the algorithmic simplifications described next.

8 Simplifying communication patterns

In [11] a very simple form of parallel quicksort for hypercubes is described. But it can also be used on other machines. In particular when P is a power of two: Simply set $P_{\text{split}} := P'/2$. A “small” PE with number i sends its large items to PE $i + P_{\text{split}}$. A “large” PE with number i sends its small items to PE $i - P_{\text{split}}$. This has the following advantages over the more complicated algorithm:

- Ideally, only half the data is moved for each level of recursion.
- The prefix sums and other collective communications are saved.
- Every PE sends and receives exactly one packet.

- The communication pattern is very regular. This effect significantly increases the effective network bandwidth.
- On a 2-D mesh, the overall distance traveled by a data item through all $\log P$ levels of recursion is in $\mathrm{O}\left(\sqrt{P}\right)$ even for row-major indexing. For row-major and shuffle indexing it is at most $2\sqrt{P}-2$, i.e., the network diameter thus matching a trivial lower bound. Since the latter two indexings additionally imply more regular communication patterns than the Hilbert indexing they are the best choice now.

The drawback of this simple approach is that the load imbalance can grow exponentially with the recursion depth. But for randomly distributed data, the median-of-medians pivot selection strategy described above is so accurate that the resulting imbalance is quite tolerable. For randomly distributed data, the imbalance even *decreases*. Apparently, the property of random data distribution is destroyed by the prefix sum based algorithm while it is maintained by the simplified algorithm. Figure 3 compares the final load imbalance and the communication expense per recursion level with and without simplified communication. (256 PEs, row-major indexing, median of medians pivot selection. For details refer to data sheets 41 and 50.)

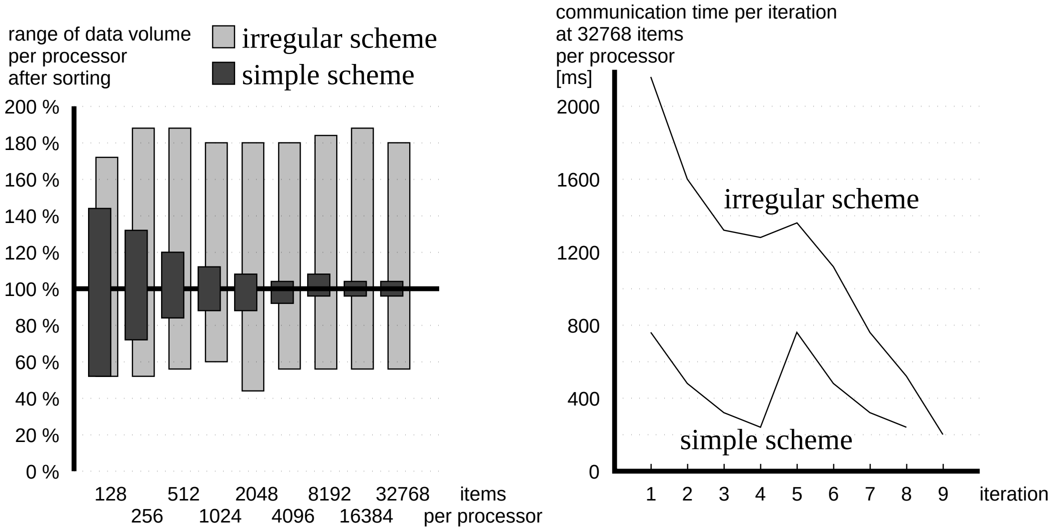


Figure 3: Final load imbalance and communication expense per recursion level with and without simplified communication for 256 PEs, row-major indexing and median of medians pivot selection. For details refer to data sheets 41 and 50.)

When the input is not randomly distributed, the simplified algorithm can still be applied by performing a random permutation of the data before starting the sorting algorithm. For large n, the additional cost for this permutation is offset by saving more than 50 % of the communication costs in the main algorithm.

9 Multiple pivots

Suppose we not only have an estimate for the median of the data items, but estimates for the $\frac{i}{k}$-quantiles[2] of the data for $0 < i < k$. Then we can split the data into k partitions using a single redistribution. For $k = P$ this technique is known as selection sort. Some of the best sorting algorithms from the theoretical point of view also use this approach with $k \ll P$ [15]. By increasing k, the recursion depth decreases but the data has to travel on longer paths and in smaller packets. Furthermore, it becomes more and more difficult to find accurate pivots.

We have implemented a multi-pivot quicksort using the medians of the local $\frac{i}{k}$-quantiles as pivots. Choosing $k = 4$ turned out to be the best choice in all cases. This value is particularly "magical" for the shuffle indexing scheme. Together with the simplified communication scheme of the previous section it turns out that the communication patterns of all iterations are identical except that the distances are halved in each iteration. While the shuffle indexing did not yield an improvement for the previously considered algorithmic variants, it now turns out to be the overall "winner". (Although the improvement is not very large in our measurements.) It implies a more regular communication pattern than the Hilbert indexing and it is superior to row-major indexing because it can exploit both horizontal and vertical network interconnections in every iteration. Figure 4 shows the execution times per item per PE on 16×16 PEs for the simplified communication pattern using $k = 2$ respectively $k = 4$. Multiple pivots yield improvements for small and medium input sizes. In the multi-pivot case, the shuffle indexing slightly outperforms row-major indexing. (For a single pivot the timings are identical for both indexing schemes. Refer to data sheets 50, 55, 66 and 69 for details.)

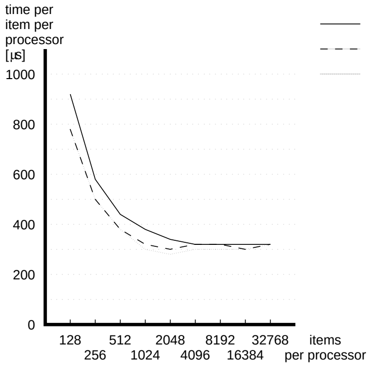


Figure 4: Execution time for different input sizes for $k = 2$ and $k = 4$. (For details refer to data sheets 50, 55, 66 and 69.)

[2] An α-quantile p_α of m items has the property that αm items are not larger than p_α and $(1 - \alpha)m$ items are not smaller than p_α.

10 Overall performance

Figure 5-(a) compares the performance of the basic parallel quicksort (no specialized routine for small numbers of PEs, median-of-3 pivot of a single PE, local sorting at the end) with the best variant we have implemented (merge-splitting sort for two PEs, initial local sorting, 3 pivots using medians of quantiles, simplified communication, shuffle indexing) on 16 × 16 PEs. The final algorithm is about three times faster than our basic algorithm. (Also compare data sheets 1 and 69). For large n it also achieves a high efficiency. Figure 5-(b) and (c) show the speedup for 256 respectively 1024 PEs.[3]

Figure 5-(d) compares our final algorithm on 32 × 32 PEs (data sheet 72) with the timings measured in [6] for five other sorting algorithms on the same machine[4] and the same measurement approach. The basic algorithm would just barely be able to compete for medium n. The optimized algorithm is the best algorithm for the entire range of n measured. For medium sized inputs it is three times faster than the best of the other algorithms.

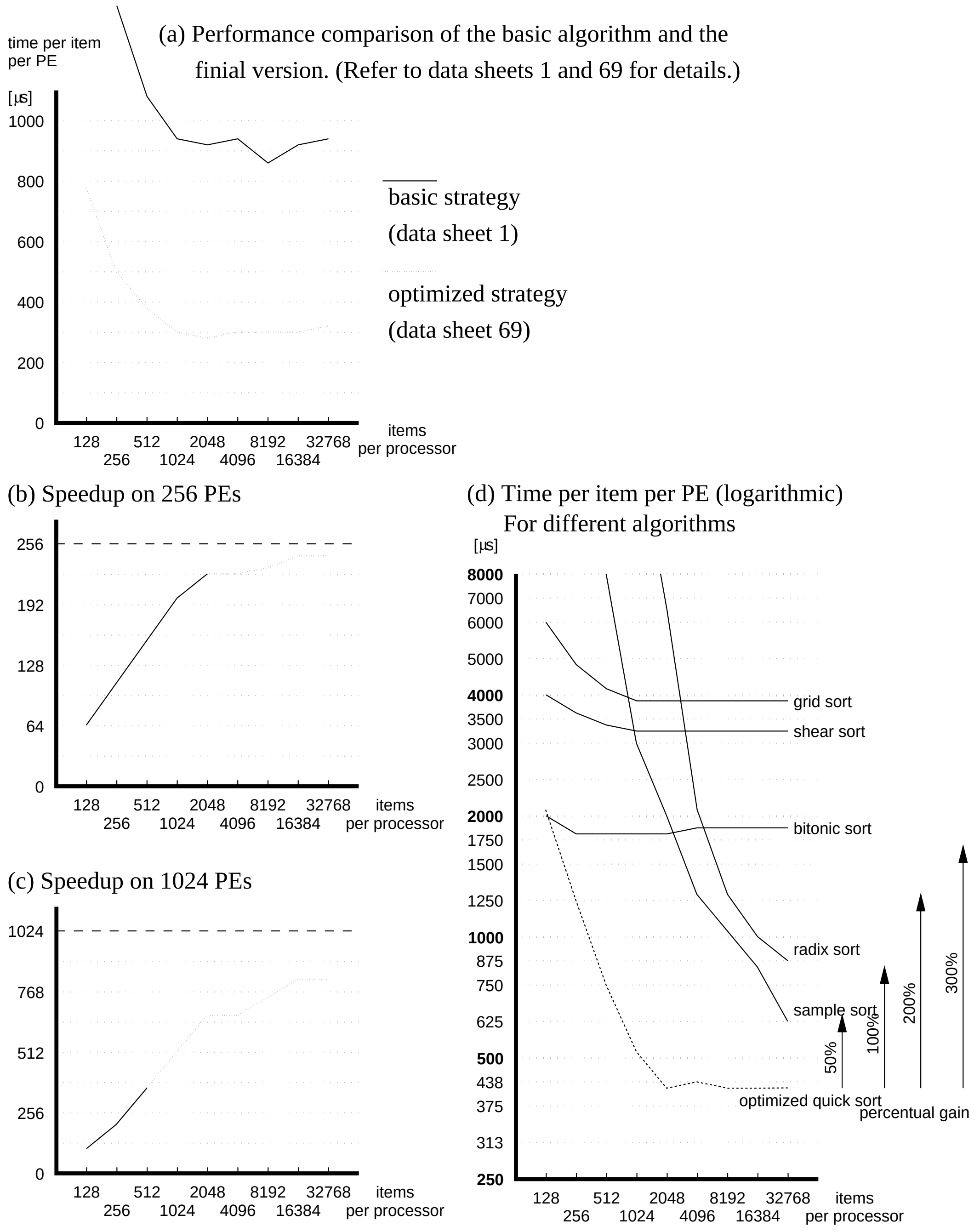
This demonstrates that quicksort has to be counted among the best practical sorting algorithms – in particular for meshes. However, it is too early to claim its superiority over the other algorithms because we do not know how careful they have been implemented. For example, for large n, sample-sort has the advantage that items have to be moved only once. Although this is not a big advantage on meshes, an implementation of sample-sort which employs a carefully tuned routine for all-to-all message exchange, should be at least as efficient as quicksort for large n.

11 Coping with more general Networks

Adapting the algorithmic measures described above for meshes with higher dimensions is straightforward. We only have to adapt the indexing schemes accordingly. The same holds for non-square meshes as long as P is a power of two. For other cases and small n, straightforward measures like concentrating (locally sorted) data in a square submesh might be adequate. But at least for large n the available computation and communication capacity should be fully exploited. Fortunately, for large numbers of randomly distributed locally sorted items, we can adapt the pivot selection strategy. Consider an $a \times b$ mesh. Split the data between four submeshes of size $\lfloor a/2 \rfloor \times \lfloor b/2 \rfloor$, $\lfloor a/2 \rfloor \times \lceil b/2 \rceil$, $\lceil a/2 \rceil \times \lfloor b/2 \rfloor$, $\lceil a/2 \rceil \times \lceil b/2 \rceil$ respectively. Choose the medians of the local $\frac{\lfloor a/2 \rfloor \lfloor b/2 \rfloor}{ab}$-quantiles, $\frac{\lfloor a/2 \rfloor \lfloor b/2 \rfloor + \lfloor a/2 \rfloor \lceil b/2 \rceil}{ab}$-quantiles and

[3] The dotted parts of the line could not be directly measured because there was not enough memory to hold the data. Therefore, the sequential execution time has been estimated by fitting a curve of the form $an + bn \log n$ to the execution time of the sequential `qsort` routine.

[4] However, the Paderborn group uses Parix rather than Cosy as a communication system. Parix has lower latencies so that our results look worse for small n. Cosy allows for a higher communication bandwidth. But even artificially halving our communication bandwidth did not change the relative performance of the algorithms.



(a) Performance comparison of the basic algorithm and the finial version. (Refer to data sheets 1 and 69 for details.)

(b) Speedup on 256 PEs

(c) Speedup on 1024 PEs

(d) Time per item per PE (logarithmic) For different algorithms

Figure 5: Overall performance of the optimized algorithms.

$\frac{\lfloor a/2\rfloor\lfloor b/2\rfloor+\lfloor a/2\rfloor\lceil b/2\rceil+\lceil a/2\rceil\lfloor b/2\rfloor}{ab}$-quantiles. Changes for higher-dimensional networks, single pivots, etc. are straightforward. The single-pivot variant of this approach somewhat resembles the QSP-2-algorithm described in [18]. However, the new algorithm avoids the large load imbalance observed for QSP-2 because of better pivots and because the recursion is stopped for small meshes. Furthermore, the QSP-2 is not a sorting algorithm in the strict sense since it does not us a fixed PE-indexing.

Even networks which are not a mesh can profit from specialized subdivision strategies. For example, a cluster computer often consists of multiple small tightly coupled parallel machines interconnected by a relatively low bandwidth network with high startup overheads. In this case, it makes sense to partition the data in such a way that all subsequent communication is within the tightly coupled subnetworks.

12 Conclusions and Future Work

Parallel quicksort is among the best parallel sorting algorithms known. From a theoretical point of view, quicksort with locality preserving indexing schemes is perhaps the simplest known algorithm for meshes which is asymptotically optimal. Unlike other asymptotically optimal algorithms, the data items have to be moved only $O(\log P)$ times. This makes the algorithm appealing from a practical point of view because message startup overheads are high on todays machines. However, certain optimizations like breaking the recursion and choosing good pivots are required for achieving high performances in terms of constant factors. Sorting locally first reduces the impact of load imbalance and simplifies pivot selection. For randomly distributed inputs, a large improvement can be achieved by *simplifying* the algorithm at the right place. These improvements are so large, that it seems worthwhile to randomly permute the data initially.

There is a large number of additional optimizations which might be interesting for future research:

- If we are willing to accept that some PEs are responsible for two data partitions, quicksort can be implemented without any imbalance at all. This possibility has been considered in [18] but the required additional communication is overestimated there for large n because a PE with a partition boundary exchanges smaller messages.

- We could use better algorithms for small P. Sample sort might be a good choice.

- Currently, the median selection strategies are implemented by gathering the required data on a single PE. For large P, parallelizing pivot selection could be an improvement. This is also a prerequisite for reconciling the theoretical analysis with the practically useful optimizations. For example, the parallel median selection algorithm described in [16] might be useful.

- Try pivot selection strategies based on random samples which do not require a random data distribution.
- Tune message exchange and merging steps such that copying is minimized.
- Currently, we use a relatively simple, high level implementation for collective communication. Exploiting tuned implementations (which are available in high quality MPI-Implementations for example) should yield significant improvements for small n.

Ultimately, the goal could be a toolbox of reusable components which can be configured to yield a very efficient sorting algorithm on many different architectures and for different input specifications. We expect that for very large n or small P, sample sort or one of its deterministic relatives (e.g. [12]) will be the method of choice. For smaller n or for sorting samples, quicksort will be better – in particular on meshes. For very small amounts of data, specialized methods like the ones used in [16] can be used. Models for the algorithmic components and the machines could be calibrated using profiling data in order to make it possible to automatically plan an optimal combination of methods for every situation.

References

[1] S. Aluru, S. Goil, and S. Ranka. Concatenated parallelism: A technique for efficient parallel divide and conquer. Technical Report SCCS-759, NPAC Syracuse University, 1996.

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, and R. Carter. The NAS parallel benchmarks. Technical Report RNR-94-007, RNR, 1994.

[3] 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 *ACM Symposium on Parallel Architectures and Algorithms*, pages 3–16, 1991.

[4] R. Butenuth, W. Burke, and H.-U. Heiß. Cosy – an operating system for highly parallel computers. *ACM Operating Systems Review*, 30(2):81–91, 1996.

[5] G. Chochia, M. Cole, and T. Heywood. Implementing the Hierarchical PRAM on the 2D mesh: Analyses and experiments. In *IEEE Symp. on Parallel and Distributed Processing*, 1995.

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

[7] T. Hansch. Sortieren großer Datenmengen auf Gittern mit Quicksort. Diplomarbeit, 1996.

[8] J. Hardwick. An efficient implementation of nested data parallelism for irregurlar divide-and-conquer algorithms. In *Workshop on High-Level Programming Models and Supportive Environments*, Honolulu, Hawaii, 1996.

[9] W. L. Hightower, J. F. Prins, and J. H. Reif. Implementations of randomized sorting on large parallel machines. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 158–167, 1992.

[10] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[11] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.

[12] H. Li and K. C. Sevcik. Parallel sorting by overpartitioning. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 46–56, Cape May, New Jersey, 1994.

[13] R. Niedermeier, K. Reinhard, and P. Sanders. Towards optimal locality in mesh-indexings. submitted for publication, 1996.

[14] R. Niedermeier and P. Sanders. On the Manhattan-distance between points on space-filling mesh-indexings. Technical Report IB 18/96, Universität Karlsruhe, Fakultät für Informatik, 1996.

[15] S. Rajasekaran and S. Sen. Random sampling techniques and parallel algorithm design. In H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 9, pages 411–451. Morgan Kaufmann, 1993.

[16] P. Sanders. Fast priority queues for parallel branch-and-bound. In *Workshop on Algorithms for Irregularly Structured Problems*, number 980 in LNCS, pages 379–393, Lyon, 1995. Springer.

[17] P. Sanders. A scalable parallel tree search library. In S. Ranka, editor, *2nd Workshop on Solving Irregular Problems on Distributed Memory Maschines*, Honolulu, Hawaii, 1996.

[18] V. Singh, V. Kumar, G. Agha, and C. Tomlinson. Efficient algorithms for parallel sorting on mesh multicomputers. *International Journal of Parallel Programming*, 20(2):95–131, 1991.

[19] T. Umland. Parallel sorting revisited. *Parallel Computing*, 20(1):115–124, 1994.

A Data sheets

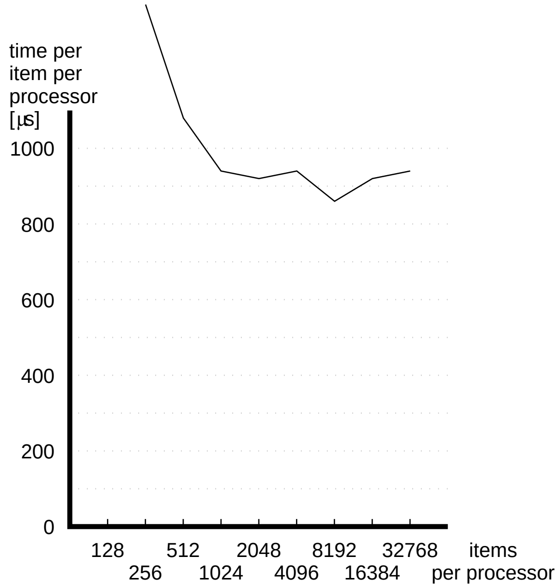
We have done extensive measurements for a wide range of algorithmic variants using the same measurement regime. This appendix shows a selection of 11 out of 72 data sheets which are particularly relevant for the points discussed here. The remaining data sheets are available on the WWW under `http://liinwww.ira.uka.de/~sanders/sorting/`. We now give a short account on the meaning of the fields in the tables. A more detailed discussion can be found in [7] (also available under the above URL). All the values shown below are averages over ten measurements. The topmost table defines the algorithmic variant used: Initial or final local sorting? *Machine* used. *Number of pivots* used. *Indexing scheme* used. (Besides the row-major, snakelike, Hilbert and shuffle indexing schemes described above, we have also experimented with variants of the latter two called H-indexing and Z-indexing.) *Pivot selection* strategy. *Reduced communication*, *exact partitioning* and *exact partners* together constitute what we have called "simplified communication pattern" here. *Data exchange inversion* stands for a variant of the data redistribution phase which yields a more regular communication pattern for the Hilbert indexing scheme.

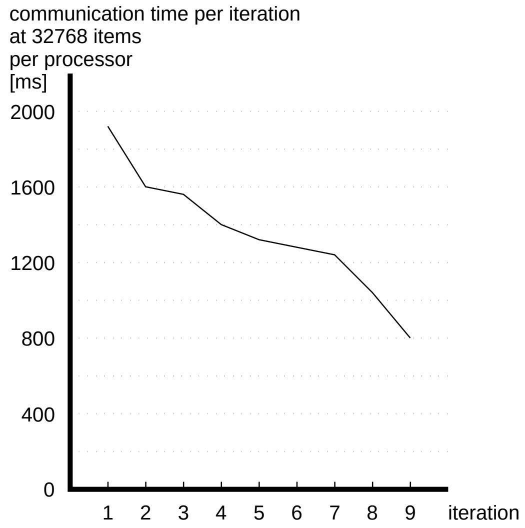
In the middle of the sheet there are four graphs showing the parallel execution time divided by n, the maximal communication expense for the i-th level of recursion, the final load balance and the relative amount of work invested in local sorting. All graphs except the second show these values for $n = 2^7$ through $n = 2^{15}$. The communication expense per recursion level is only shown for $n = 2^{15}$.

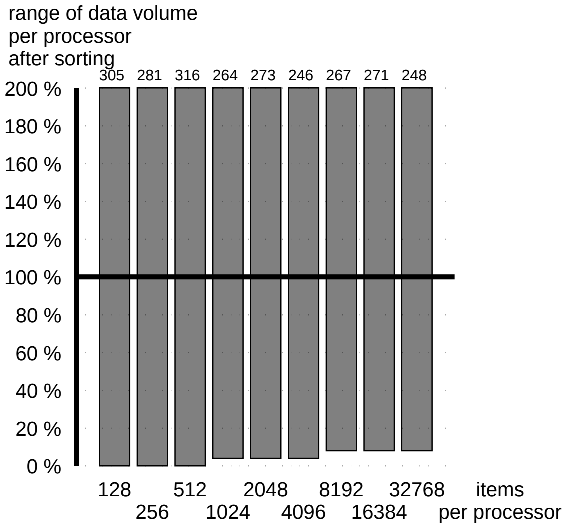
The table at the bottom of the sheet shows numeric measurements for the different input sizes. Namely, parallel execution time divided by n, parallel execution time, time for the Quicksort recursion, time for local sorting, time for merge splitting sort, minimum and maximum number of items ending up on any PE, number of network packets sent, total path-length of all network packets and average recursion depth reached.

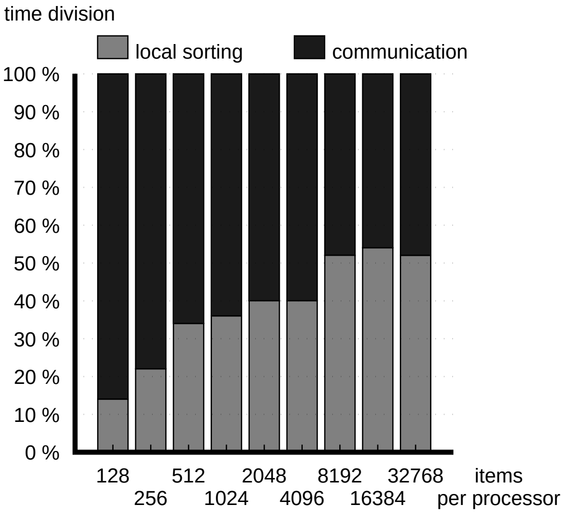
Data Sheet 1

Quicksort	with final local sorting		
machine	GCel with 256 processors		
number of pivots	1	**processor numbering**	row major
pivot selection	3-median of root processor		
reduced communication	no	**exact bisection**	no
exact partner processors	no	**data exchange inversion**	no





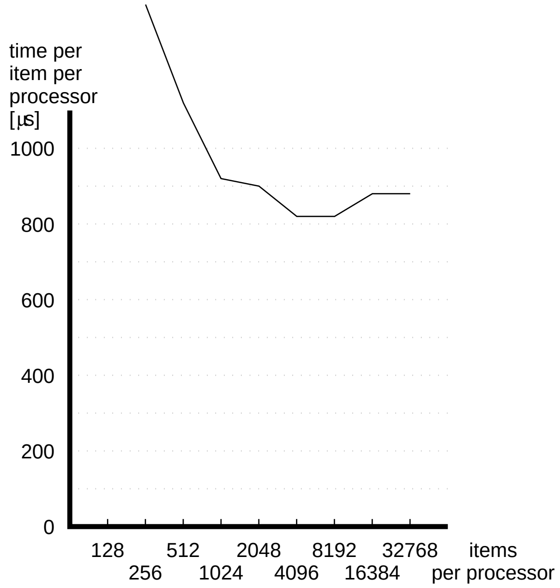


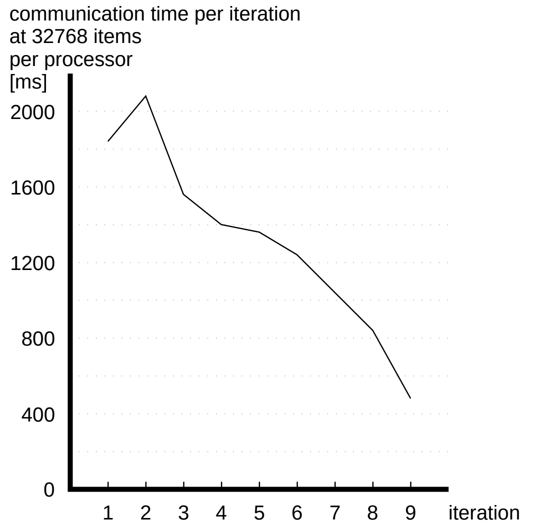


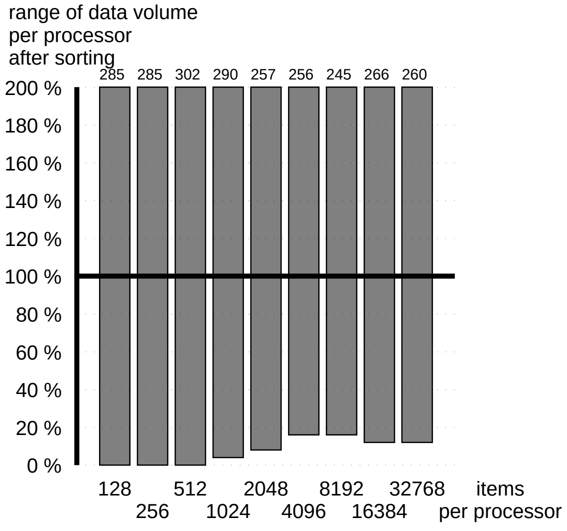
items per processor	128	256	512	1024	2048	4096	8192	16384	32768
time per item per processor [μs]	1976	1378	1081	944	924	942	859	917	931
total time [s]	0.25	0.35	0.55	0.97	1.89	3.86	7.05	15.04	30.54
time for communication [ms]	232	294	396	671	1344	2506	3856	7553	16362
time for local sorting [ms]	39	83	212	382	880	1689	4049	8813	17231
time for odd-even sort [ms]	0	0	0	0	0	0	0	0	0
minimum number of items	0	2	3	29	49	136	516	806	2682
maximal number of items	391	720	1620	2704	5588	10090	21900	44475	81214
number of data packets	6849	7167	6756	6827	9049	13920	23457	41581	84270
total path length of all data packets	34608	36224	34628	35120	46604	67850	114075	196278	409104
average recursion depth	9.62	10.10	9.48	9.51	9.82	9.74	9.48	9.07	9.60

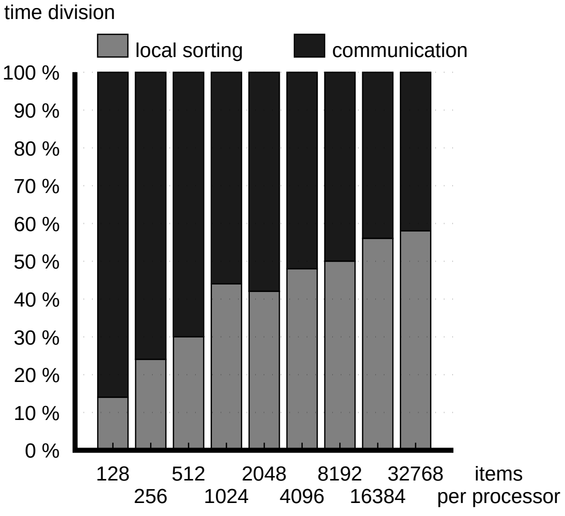
Data Sheet 2

Quicksort	with final local sorting		
machine	GCel with 256 processors		
number of pivots	1	**processor numbering**	snake-like
pivot selection	3-median of root processor		
reduced communication	no	**exact bisection**	no
exact partner processors	no	**data exchange inversion**	no





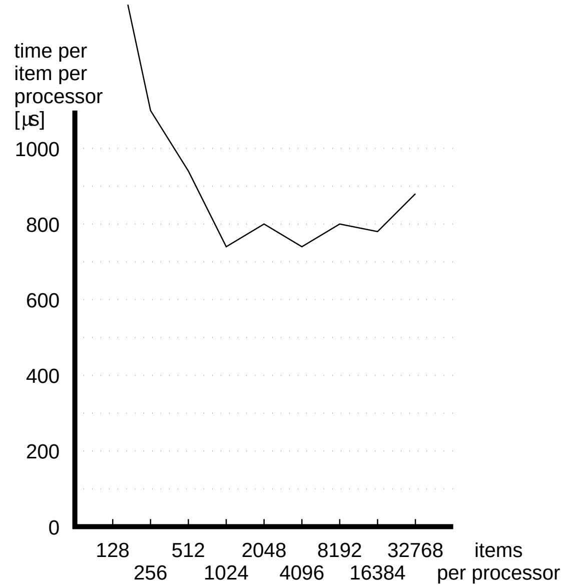


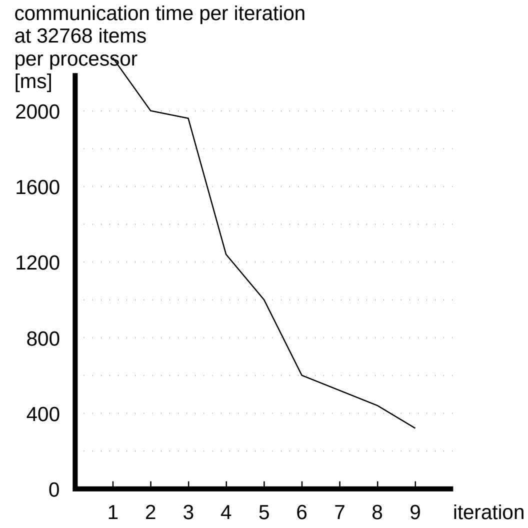


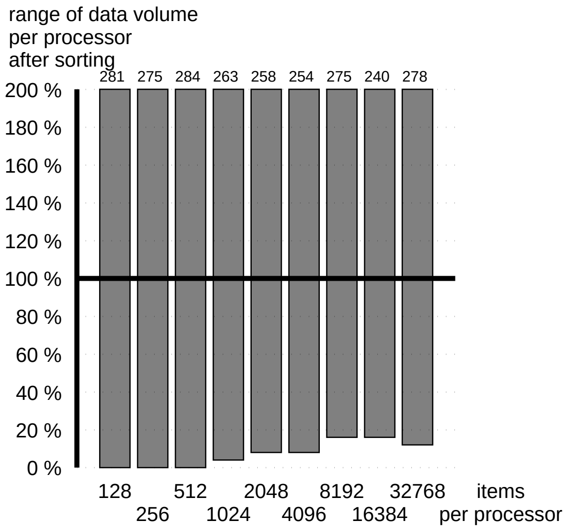
items per processor	128	256	512	1024	2048	4096	8192	16384	32768
time per item per processor [μs]	2077	1374	1159	913	903	815	830	878	887
total time [s]	0.27	0.35	0.59	0.94	1.85	3.34	6.80	14.39	29.10
time for communication [ms]	240	283	459	575	1313	1979	3821	6976	13645
time for local sorting [ms]	38	83	200	426	825	1780	3691	8563	18149
time for odd-even sort [ms]	0	0	0	0	0	0	0	0	0
minimum number of items	1	0	4	21	74	342	648	853	2049
maximal number of items	366	731	1545	2970	5280	10510	20139	43670	85506
number of data packets	6804	7286	7042	6836	8896	13302	23046	40781	80270
total path length of all data packets	30809	32701	32638	31290	41059	57847	98459	171073	337330
average recursion depth	9.59	10.25	9.85	9.51	9.70	9.40	9.36	8.92	9.17

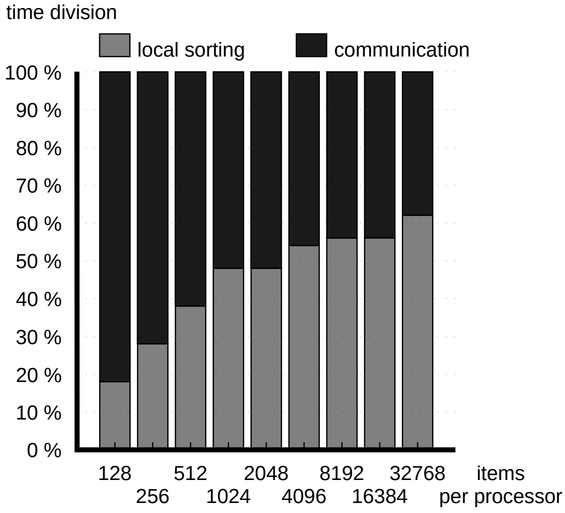
Data Sheet 3

Quicksort	with final local sorting		
machine	GCel with 256 processors		
number of pivots	1	**processor numbering**	Hilbert numbering
pivot selection	3-median of root processor		
reduced communication	no	**exact bisection**	no
exact partner processors	no	**data exchange inversion**	no





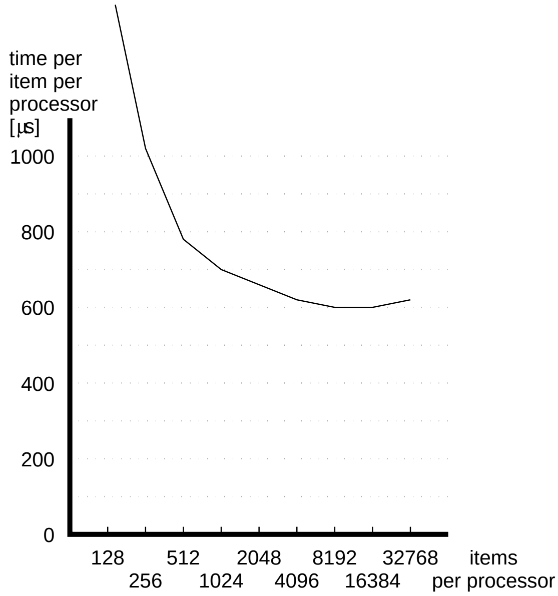


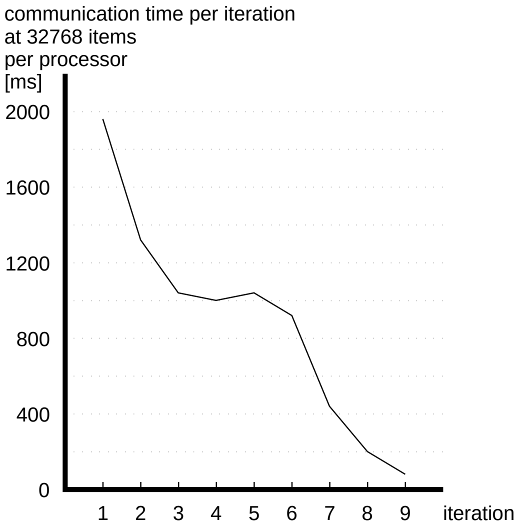


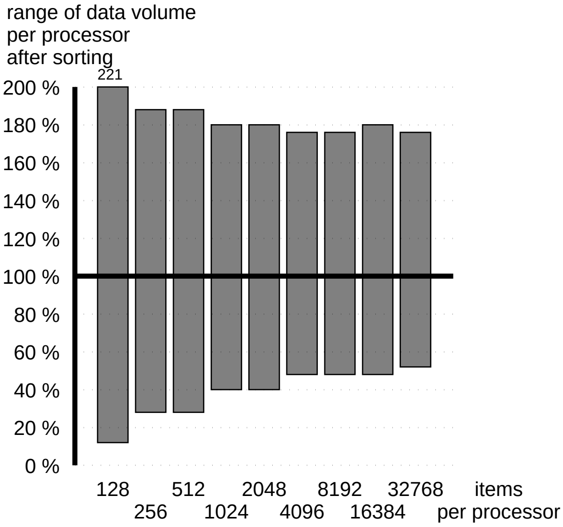
items per processor	128	256	512	1024	2048	4096	8192	16384	32768
time per item per processor [μs]	1577	1096	935	729	791	741	804	790	888
total time [s]	0.20	0.28	0.48	0.75	1.62	3.04	6.59	12.96	29.13
time for communication [ms]	176	211	320	436	922	1549	3248	6186	12417
time for local sorting [ms]	38	82	192	383	830	1749	4191	7681	19263
time for odd-even sort [ms]	0	0	0	0	0	0	0	0	0
minimum number of items	0	1	6	20	76	147	617	1238	1932
maximal number of items	360	705	1456	2698	5277	10405	22497	39281	90938
number of data packets	6891	7122	6722	6960	8844	13339	23220	42702	82651
total path length of all data packets	27368	28037	27440	27985	34109	49197	84703	154494	296643
average recursion depth	9.68	10.03	9.43	9.68	9.64	9.42	9.41	9.34	9.41

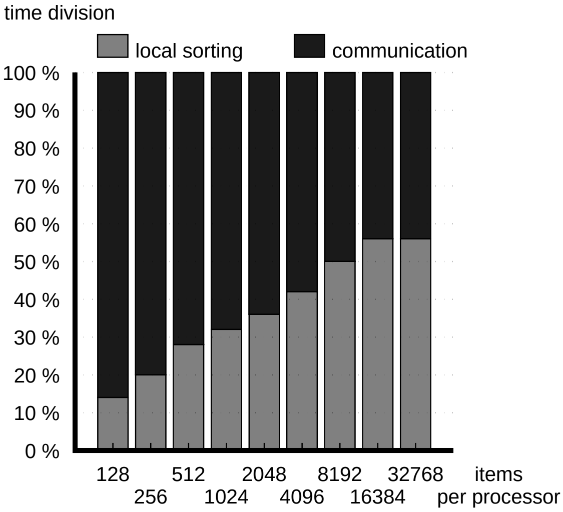
Data Sheet 12

Quicksort	with final local sorting		
machine	GCel with 256 processors		
number of pivots	1	**processor numbering**	row major
pivot selection	Median of $\sqrt{N}$-medians		
reduced communication	no	**exact bisection**	no
exact partner processors	no	**data exchange inversion**	no





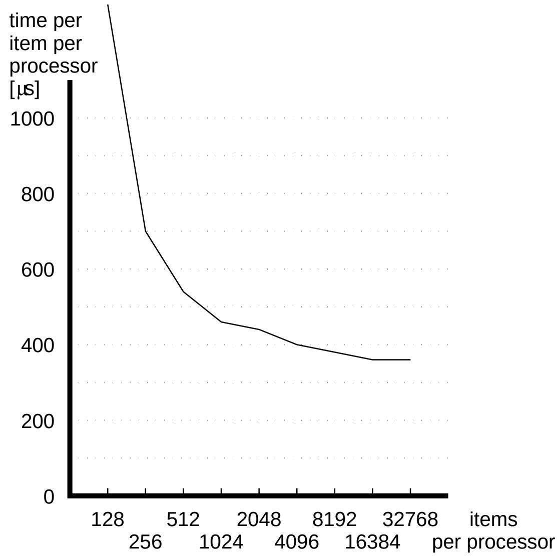


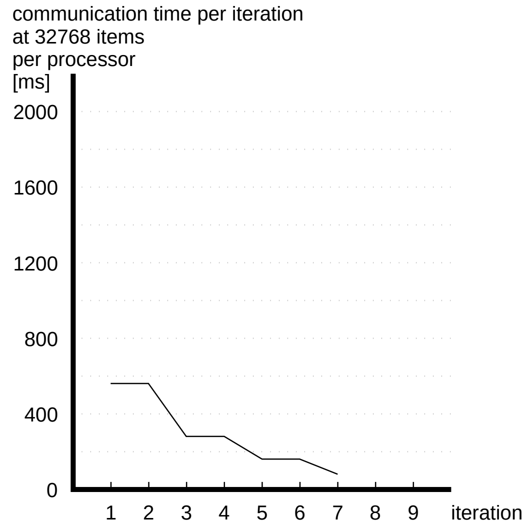


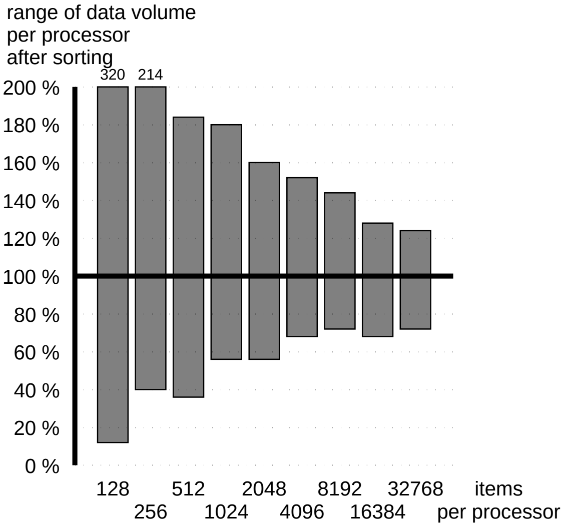
items per processor	128	256	512	1024	2048	4096	8192	16384	32768
time per item per processor [μs]	1452	1022	784	695	663	614	597	610	611
total time [s]	0.19	0.26	0.40	0.71	1.36	2.52	4.90	10.04	20.05
time for communication [ms]	168	218	305	517	945	1571	2581	4661	9237
time for local sorting [ms]	29	52	117	248	547	1172	2552	5705	11727
time for odd-even sort [ms]	0	0	0	0	0	0	0	0	0
minimum number of items	17	74	151	426	840	1919	3813	8157	16812
maximal number of items	284	483	959	1855	3682	7267	14469	29788	57356
number of data packets	5764	5692	5706	5670	7278	11655	20062	36818	70216
total path length of all data packets	29686	30942	30967	30143	38225	57784	96999	164197	330664
average recursion depth	8.20	8.10	8.11	8.06	8.06	8.03	8.03	8.02	8.01

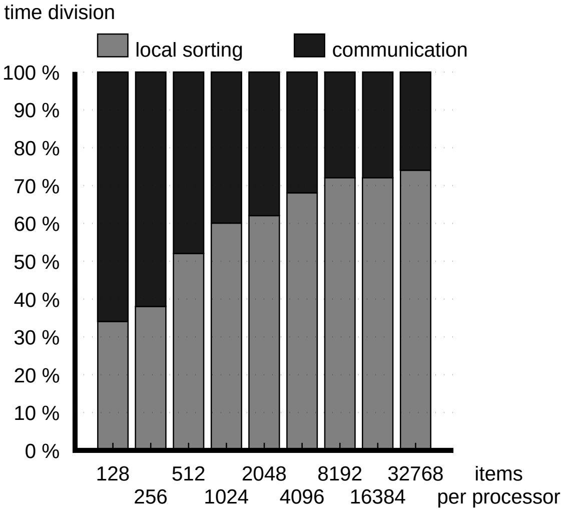
Data Sheet 37

Quicksort	with final local sorting and odd-even sort for 2 Proc.		
machine	GCel with 256 processors		
number of pivots	1	**processor numbering**	Shuffle numbering
pivot selection	Median of $\sqrt{N}$-medians		
reduced communication	yes	**exact bisection**	yes
exact partner processors	yes	**data exchange inversion**	no





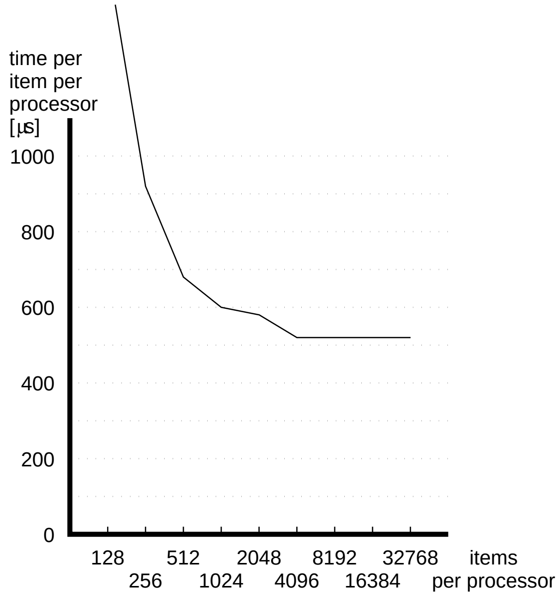


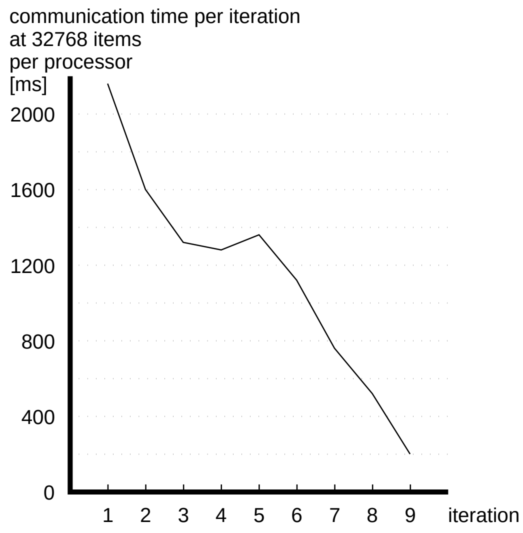


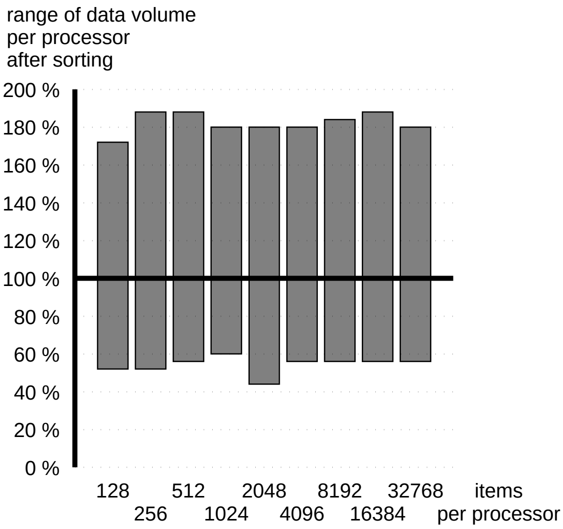
items per processor	128	256	512	1024	2048	4096	8192	16384	32768
time per item per processor [μs]	1108	706	548	460	439	406	390	351	370
total time [s]	0.14	0.18	0.28	0.47	0.90	1.67	3.20	5.76	12.14
time for communication [ms]	100	110	140	194	350	548	900	1654	3092
time for local sorting [ms]	41	60	129	250	489	1019	2094	3756	8363
time for odd-even sort [ms]	10	10	20	32	60	111	208	363	722
minimum number of items	17	102	180	570	1133	2780	5875	11357	23794
maximal number of items	410	550	952	1829	3246	6220	11919	20796	41102
number of data packets	2048	2048	2048	2186	3475	5890	10616	20031	38765
total path length of all data packets	7680	7680	7680	7818	12527	21117	37376	69830	133696
average recursion depth	7.00	7.00	7.00	7.00	7.00	7.00	7.00	7.00	7.00

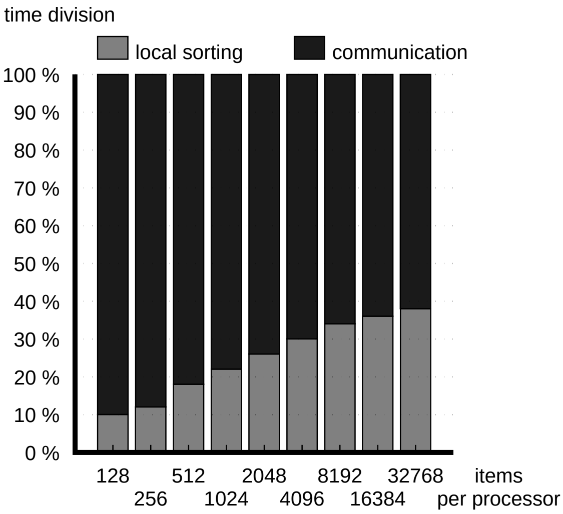
Data Sheet 41

Quicksort	with initial local sorting		
machine	GCel with 256 processors		
number of pivots	1	**processor numbering**	row major
pivot selection	Median of medians		
reduced communication	no	**exact bisection**	no
exact partner processors	no	**data exchange inversion**	no





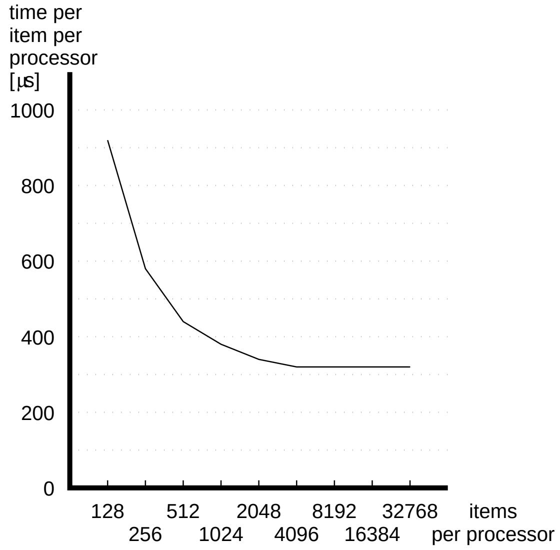


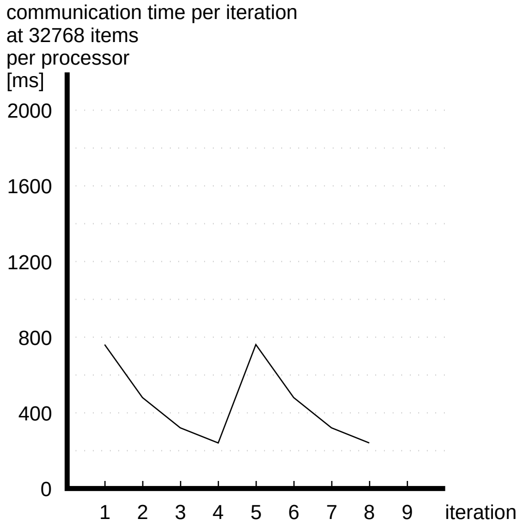


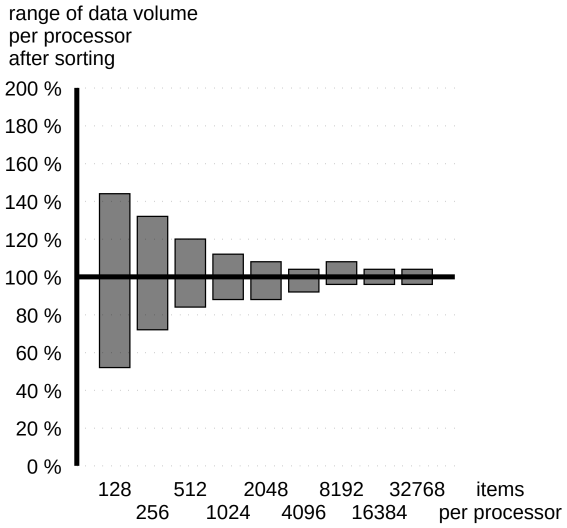
items per processor	128	256	512	1024	2048	4096	8192	16384	32768
time per item per processor [μs]	1406	913	671	608	585	515	522	515	525
total time [s]	0.18	0.23	0.34	0.62	1.20	2.11	4.28	8.45	17.22
time for communication [ms]	171	220	300	506	940	1525	3005	5669	11237
time for local sorting [ms]	19	30	69	140	319	670	1524	3182	6905
minimum number of items	66	135	284	607	938	2319	4612	9216	18345
maximal number of items	218	485	963	1845	3696	7433	14987	31081	58893
number of data packets	5617	5639	5637	5642	7340	11493	19777	36510	69962
total path length of all data packets	26799	26039	24244	25567	31968	47915	83747	149185	284578
average recursion depth	8.01	8.03	8.02	8.02	8.02	8.01	8.02	8.02	8.03

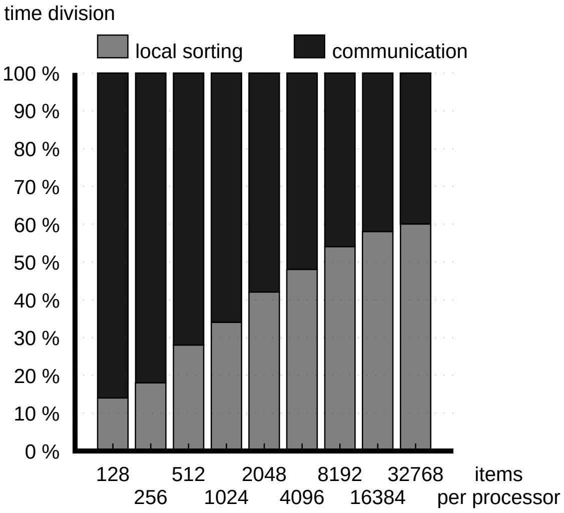
Data Sheet 50

Quicksort	with initial local sorting		
machine	GCel with 256 processors		
number of pivots	1	**processor numbering**	row major
pivot selection	Median of medians		
reduced communication	yes	**exact bisection**	yes
exact partner processors	yes	**data exchange inversion**	no





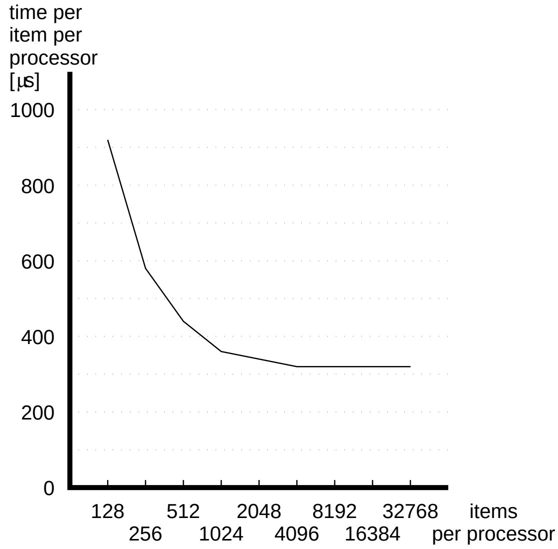


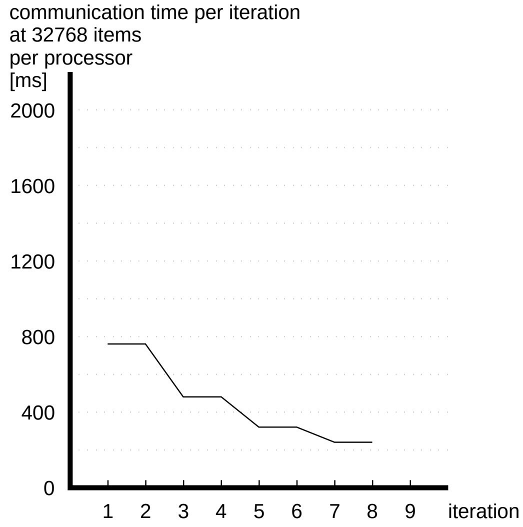


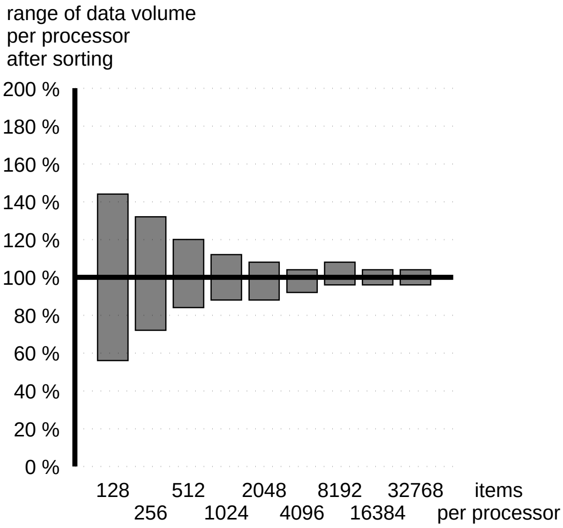
items per processor	128	256	512	1024	2048	4096	8192	16384	32768
time per item per processor [μs]	929	585	449	371	345	312	313	311	322
total time [s]	0.12	0.15	0.23	0.38	0.71	1.28	2.57	5.11	10.58
time for communication [ms]	114	131	180	269	448	699	1299	2327	4591
time for local sorting [ms]	18	30	69	141	314	670	1524	3181	6906
minimum number of items	71	188	420	905	1835	3784	7869	15871	31905
maximal number of items	182	337	618	1161	2232	4330	8767	16945	33500
number of data packets	2048	2048	2048	2048	3511	5705	9988	18303	34775
total path length of all data packets	7680	7680	7680	7680	13300	21521	37530	68717	130452
average recursion depth	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00

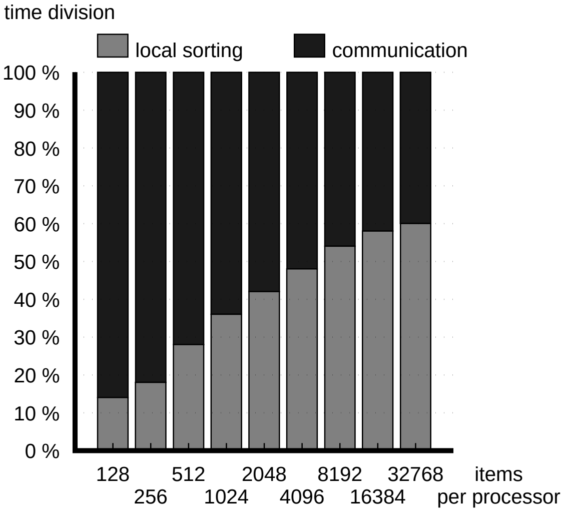
Data Sheet 55

Quicksort	with initial local sorting		
machine	GCel with 256 processors		
number of pivots	1	**processor numbering**	Shuffle numbering
pivot selection	Median of medians		
reduced communication	yes	**exact bisection**	yes
exact partner processors	yes	**data exchange inversion**	no





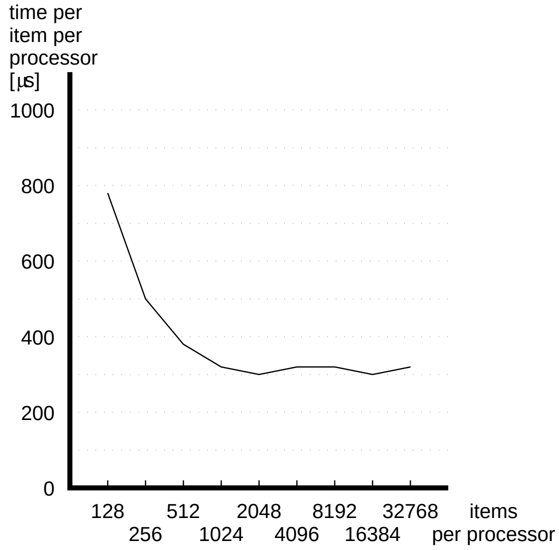


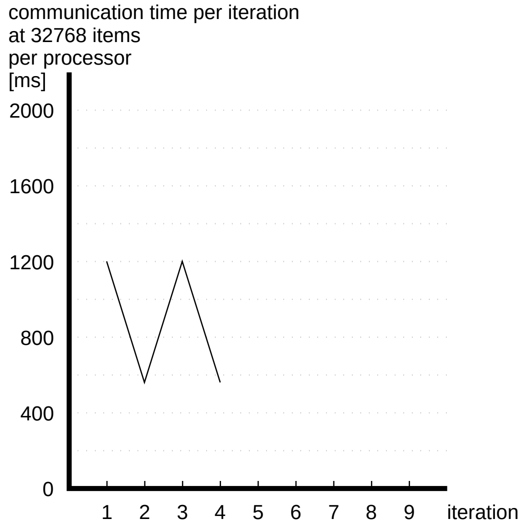


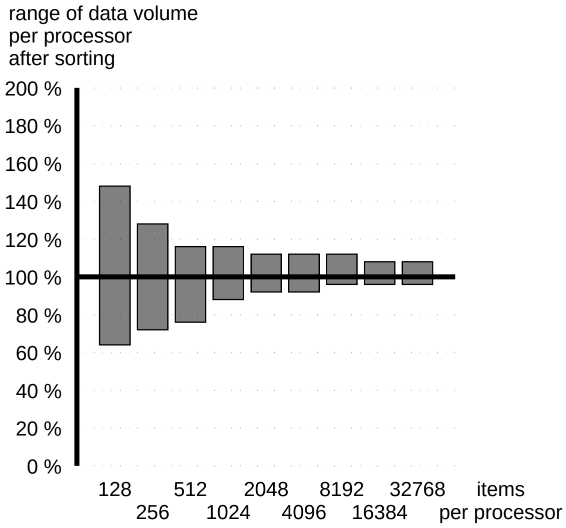
items per processor	128	256	512	1024	2048	4096	8192	16384	32768
time per item per processor [μs]	913	585	449	365	342	312	313	311	322
total time [s]	0.12	0.15	0.23	0.37	0.70	1.28	2.57	5.10	10.59
time for communication [ms]	110	130	180	260	446	694	1294	2322	4604
time for local sorting [ms]	19	30	69	141	319	670	1524	3182	6908
minimum number of items	71	188	420	905	1835	3784	7869	15871	31905
maximal number of items	182	337	618	1161	2232	4330	8767	16945	33500
number of data packets	2048	2048	2048	2048	3511	5705	9988	18303	34775
total path length of all data packets	7680	7680	7680	7680	13397	21586	37652	68797	130471
average recursion depth	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00	8.00

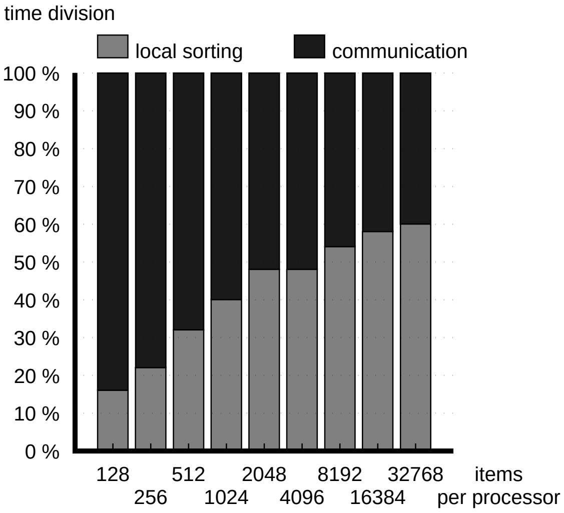
Data Sheet 66

Quicksort	with initial local sorting		
machine	GCel with 256 processors		
number of pivots	3	**processor numbering**	row major
pivot selection	multi-median of pivot candidates		
reduced communication	yes	**exact partioning**	yes
exact partner processors	yes	**data exchange inversion**	yes





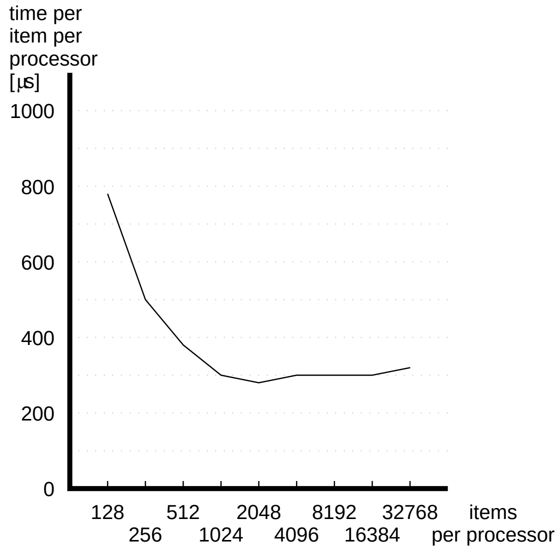


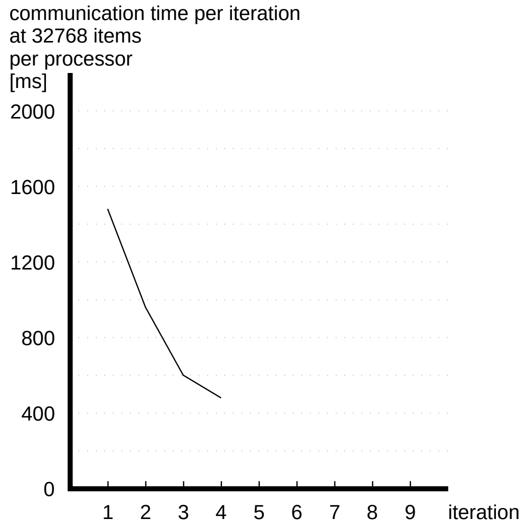


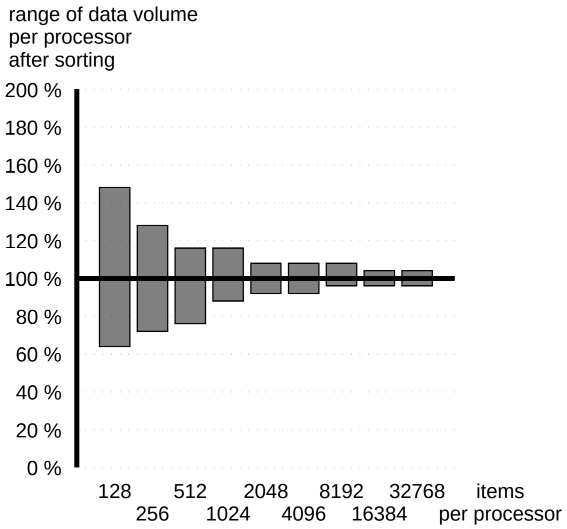
items per processor	128	256	512	1024	2048	4096	8192	16384	32768
time per item per processor [μs]	781	507	386	312	292	314	313	310	320
total time [s]	0.10	0.13	0.20	0.32	0.60	1.29	2.57	5.08	10.50
time for communication [ms]	18	110	150	210	340	707	1298	2301	4505
time for local sorting [ms]	100	30	69	142	314	670	1526	3181	6904
minimum number of items	81	181	388	913	1884	3818	7781	15866	31960
maximal number of items	190	330	603	1188	2248	4394	8711	16982	33560
number of data packets	4096	4096	4096	4096	4096	7019	11421	19919	36600
total path length of all data packets	12800	12800	12800	12800	12800	22121	35928	62385	114410
average recursion depth	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

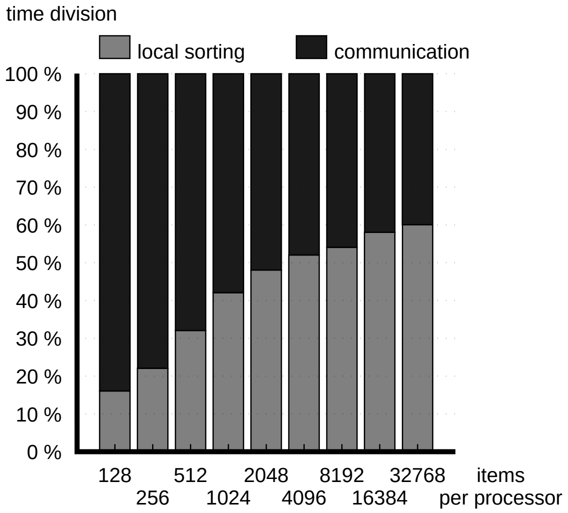
Data Sheet 69

Quicksort	with initial local sorting		
machine	GCel with 256 processors		
number of pivots	3	**processor numbering**	Shuffle numbering
pivot selection	multi-median of pivot candidates		
reduced communication	yes	**exact partioning**	yes
exact partner processors	yes	**data exchange inversion**	no





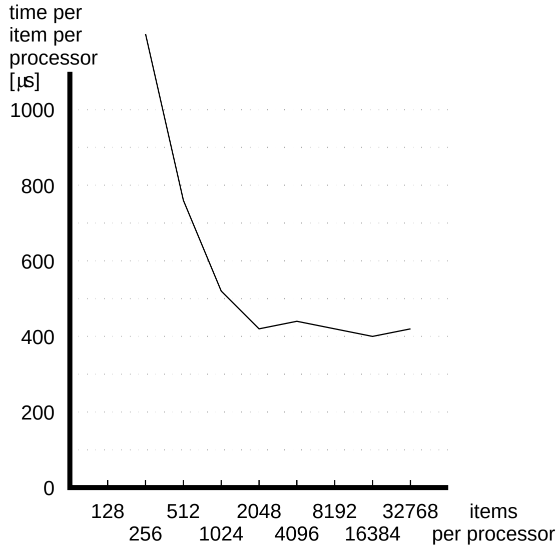


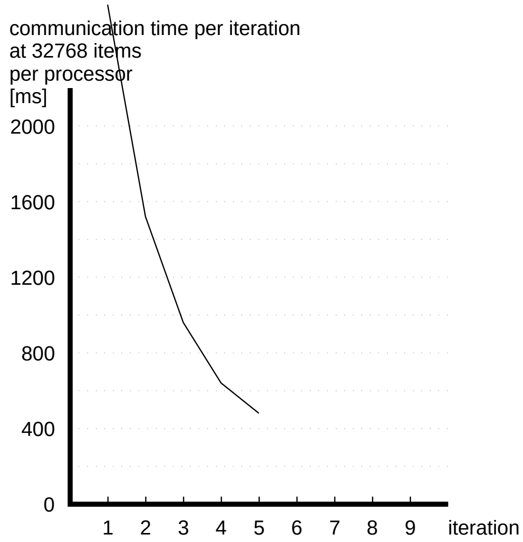


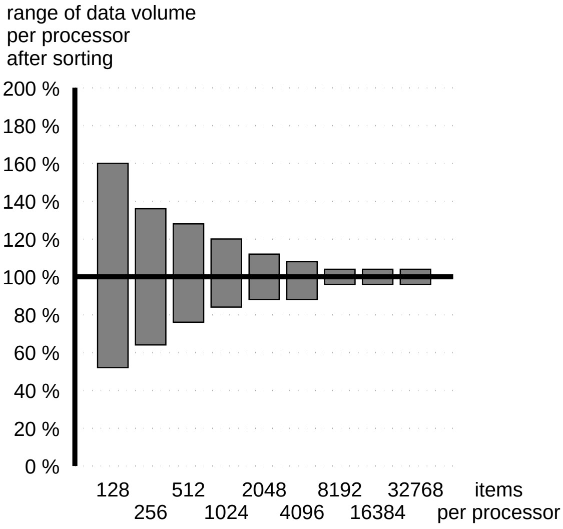
items per processor	128	256	512	1024	2048	4096	8192	16384	32768
time per item per processor [μs]	781	503	372	302	283	303	307	308	321
total time [s]	0.10	0.13	0.19	0.31	0.58	1.24	2.53	5.06	10.54
time for communication [ms]	100	110	141	200	318	658	1250	2274	4555
time for local sorting [ms]	19	30	69	140	317	670	1526	3181	6904
minimum number of items	81	181	388	913	1884	3818	7780	15866	31959
maximal number of items	190	330	603	1188	2248	4394	8711	16982	33560
number of data packets	4096	4096	4096	4096	4096	7019	11421	19919	36600
total path length of all data packets	15360	15360	15360	15360	15360	26790	43506	75266	137601
average recursion depth	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00	4.00

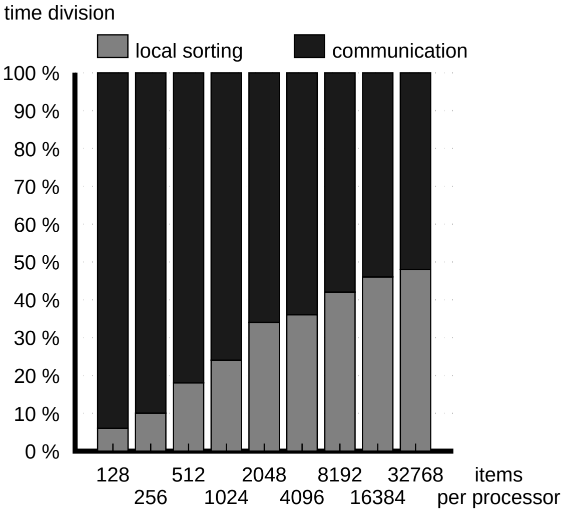
Data Sheet 72

Quicksort	with initial local sorting		
machine	GCel with 1024 processors		
number of pivots	3	**processor numbering**	Shuffle numbering
pivot selection	multi-median of pivot candidates		
reduced communication	yes	**exact partioning**	yes
exact partner processors	yes	**data exchange inversion**	no









items per processor	128	256	512	1024	2048	4096	8192	16384	32768
time per item per processor [μs]	2093	1210	759	527	427	445	421	402	412
total time [s]	0.27	0.31	0.39	0.54	0.88	1.83	3.45	6.60	13.53
time for communication [ms]	269	290	340	422	616	1247	2182	3819	7595
time for local sorting [ms]	19	30	75	140	323	703	1526	3223	7123
minimum number of items	66	165	392	878	1803	3644	7742	15581	31892
maximal number of items	204	346	636	1221	2279	4457	8657	17036	33889
number of data packets	20480	20480	20480	20480	20480	34894	56920	99435	182799
total path length of all data packets	126976	126976	126976	126976	126976	221504	359614	622514	1137188
average recursion depth	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00