

Parallel sorting algorithms for tightly coupled multiprocessors

Michael J. QUINN

Parallel Computing Laboratory, Department of Computer Science, University of New Hampshire, Durham, NH 03824, U.S.A.

Received November 1986

Abstract. We present three parallel sorting algorithms suitable for implementation on tightly coupled multiprocessors and compare their performance on the Denelcor HEP. Two of the algorithms implemented—parallel Shellsort and quickmerge—are new. Shellsort is amenable to parallelization; however, since Shellsort has higher complexity than quicksort, parallel Shellsort is inferior to parallel quicksort. A second new parallel algorithm, called *quickmerge*, is based upon both quicksort and mergesort. Our implementation of quickmerge achieves significantly higher speedup than our implementation of parallel quicksort.

Keywords. Shared memory multiprocessor, Denelcor HEP, sorting algorithms, parallel Shellsort, quickmerge, parallel quicksort, timing results.

1. Introduction

Sorting is one of the most common activities performed on computers. Many algorithms incorporate a sort so that later accesses of information may be done efficiently. This paper describes three sorting algorithms designed for tightly coupled multiprocessors. All of these algorithms are internal sorts—that is, they sort tables small enough to fit entirely in primary memory. In addition, all of the algorithms described below sort by comparing pairs of keys.

The tightly coupled multiprocessor model consists of a number of fully programmable processors, each capable of executing its own set of instructions. Every processor has a private memory for storing its instructions and local data. The processors access a shared global memory through a central switching mechanism. Examples of tightly coupled multiprocessors include the Denelcor HEP [6], the Encore Multimax, the Sequent Balance 8000 [11], and New York University's Ultracomputer [3]. See [8] for more detailed descriptions of this and other models of parallel computation.

Little work has been reported on multiprocessor sorting algorithms. Raskin implemented a parallel quicksort algorithm for the Cm*, a loosely coupled multiprocessor. Later Deminet improved Raskin's algorithm. These parallel algorithms allow multiple unsorted sublists to be partitioned concurrently. The parallel quicksort algorithm described in Section 2 is essentially identical to Deminet's algorithm.

Recently Halstead has implemented a parallel quicksort algorithm in Multilisp on Concert, an experimental tightly coupled multiprocessor [4]. Multilisp constructs allow processes to begin partitioning an unsorted sublist even before that sublist has been completely generated by another partitioning process. Halstead's algorithm achieves nearly linear speedup on the Concert for small numbers of processors. However, his algorithm contains a sequential component—the partitioning of the original list—that puts a $\Omega(n)$ lower bound on the complexity of the sort, even as the number of processors becomes arbitrarily large.

Our motivation is to find a parallel sorting algorithm that produces nearly linear speedup in a more traditional multiprocessor programming environment. In the following sections we describe parallel quicksort and two new algorithms and compare their performance on a single-processor Denelcor HEP. A single-processor HEP supports multiple concurrent processes through pipelining [6,8].

2. Parallel Quicksort

Quicksort, invented by Hoare [5], is a commonly-used sorting algorithm on serial computers. Its popularity is due to its good average-case behavior: given a random permutation of n keys, quicksort can be expected to sort the keys in about $2n \log n$ comparisons [1].

Let us review how the algorithm works. Quicksort is a recursive algorithm that partitions an unsorted list into two sublists and a splitter key, then sorts the sublists independently. One of these sublists contains keys less than the splitter key; the other sublist contains keys greater than the splitter key. The splitter key, located between the two sublists, is in its correctly sorted position, since it is greater than all the keys in one sublist and smaller than all the keys in the other sublist. Given an initially unsorted list, then, the quicksort algorithm chooses one element as the splitter (e.g., the first element). After a single partitioning step, the list is divided into two sublists, and the algorithm recursively partitions each of the sublists. Quicksort is an example of a divide-and-conquer algorithm: once a list has been partitioned, the two unsorted sublists form independent problems that can be solved simultaneously.

The following parallel quicksort algorithm was suggested by Sedgewick [10] and implemented by Raskin [9] and Deminet [2]. A number of processes, one per processor, execute in parallel. The keys to be sorted are stored in an array. A single global stack stores the indices of subarrays that are still unsorted. When a process is without work, it checks to see if the stack is nonempty. If the stack is nonempty, the process locks the stack, pops the indices of an unsorted subarray off the stack, and unlocks the stack. The process partitions the subarray, using the partitioning algorithm described above, into two smaller arrays, containing keys less than or greater than the splitter key. After the partitioning step, the process locks the global stack, pushes the indices for one of the unsorted subarrays onto the stack, unlocks the stack, and repeats the partitioning process on the other unsorted subarray.

We have implemented this algorithm in HEP FORTRAN on a single-PEM Denelcor HEP running under the HEP/UPX operating system. Two common modifications were made to improve running time. First, we used the median of the first, middle, and last elements of the subarray as the splitter key. Second, we sorted subarrays with fewer than ten keys using linear insertion sort.

The speed of the parallel algorithm was tested by sorting arrays of one hundred, one thousand, and ten thousand elements. We generated ten arrays of each size. Every array on n elements consisted of a random permutation of the first n integers, generated using the shuffling algorithm [7]. Figure 1 illustrates the average speedup by this parallel algorithm on the three list sizes, as a function of the number of processes.

Depending upon the memory reference patterns, the maximum speedup achievable on a single-processor HEP is between 8 and 12. As Fig. 1 makes clear, the parallel quicksort program achieves rather mediocre speedup. A mathematical analysis of the parallel quicksort algorithm yields a predicted speedup curve nearly identical to our empirical result, showing that the weakness is in the algorithm, not the implementation [8]. What constrains the speedup achieved by this parallel algorithm? The most important factor is the low amount of parallelism early in the algorithm's execution. The initial partitioning step, during which the array is partitioned into two subarrays, is performed by a single process. The other processes remain

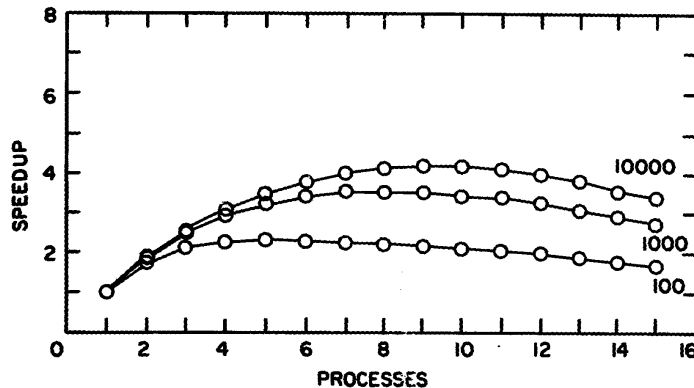


Fig. 1. Speedup of parallel quicksort algorithm on Denelcor HEP.

idle. After the array has been divided into two subarrays, two processes can each partition one of these subarrays, but again the rest of the processes have nothing to do. This is not an insignificant amount of time to wait. In fact, the partitioning of the original unsorted array forms a significant sequential component that puts a ceiling on the maximum speedup achievable, regardless of the number of processors available. A less significant factor is the contention of processes for the shared global stack containing indices of unsorted subarrays.

To summarize, the parallel quicksort algorithm has no or low parallelism at the beginning of its execution and high parallelism at the end of its execution. Is there another sequential algorithm that is more amenable to parallelization?

3. Parallel Shellsort

Shellsort is an extension of linear insertion sort that allows keys to be moved a large distance in the list after a single comparison. Execution of the algorithm consists of a number of iterations. For each iteration i there is an associated distance d_i . Elements whose indices modulo d_i are equal are sorted using linear insertion sort. Every iteration the distance is reduced. The value of d_1 , the distance on the final iteration, is 1; after this iteration the list is sorted. The complexity of the Shellsort algorithm depends upon the values chosen for the d_i 's; if $d_i = 2^i - 1$ for $1 \leq i \leq \lceil \log n \rceil$, then the worst-case time complexity of the algorithm is $O(n^{3/2})$. Figure 2 illustrates the execution of the Shellsort algorithm with a simple example.

When the distance is large, Shellsort is quite amenable to parallelization. On the first few iterations there are far more sublists to sort than processors to sort them. The work is easily partitioned by allocating each processor its fair share of the sublists.

What happens when the values of d_i become less than the number of processors? Consider the situation on the last iteration when the increment is 1. During this, the last, iteration, the algorithm performs a linear insertion sort on the list as a whole. However, because of the work done by the previous iterations of the algorithm, most elements are very close to their final position. We can take advantage of this and apply all the processors to the sort. Given p processors, each processor sorts a contiguous sublist of approximately n/p keys (Fig. 3(a)). After this step, each sublist is sorted, but there may be irregularities at the borders between two processors' areas. These discontinuities can be eliminated by sorting $p - 1$ contiguous sublists in parallel, beginning with the $n/(2p)$ th element in the list (Fig. 3(b)). Once again discontinui-

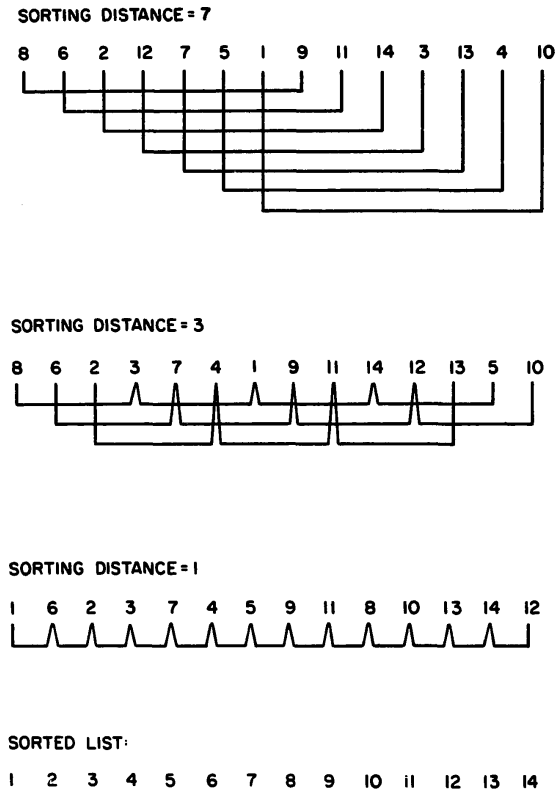


Fig. 2. Example of Shellsort algorithm.

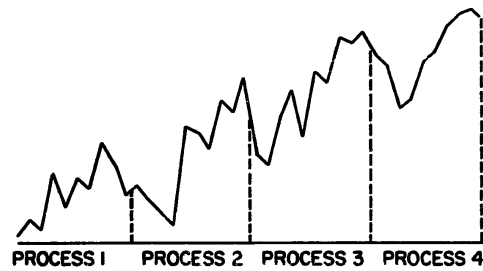
ties may appear between two processors' areas (Fig. 3(c)). Therefore, these two steps are alternately executed until the list is completely sorted.

To summarize, when there are at least as many sublists as processors, each processor sorts its fair share of sublists. When there are fewer sublists than processors, every sublist is sorted in parallel using the technique described in the previous paragraph.

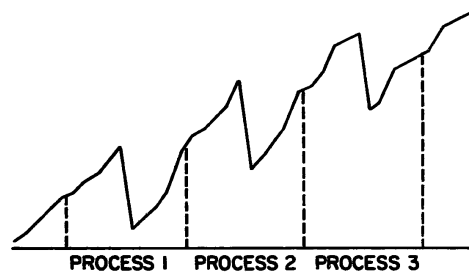
We have implemented this algorithm in HEP FORTRAN on a single-PEM Denelcor HEP running under the HEP/UPX operating system. The program sorted the same set of arrays as the parallel quicksort algorithm. The ratio of the execution time of the sequential Shellsort algorithm to this parallel Shellsort algorithm for various numbers of processes is shown in Fig. 4. Notice that the curve reflects a good utilization of processors.

Unfortunately, a fair analysis of any parallel algorithm requires that the speed of the parallel algorithm be compared with the execution time of the fastest sequential algorithm. The sequential Shellsort algorithm is several times slower than the sequential quicksort algorithm for the problem sizes we tested. Hence the speedup of the parallel shellsort algorithm is poor, reflecting its foundation on a less efficient sequential algorithm.

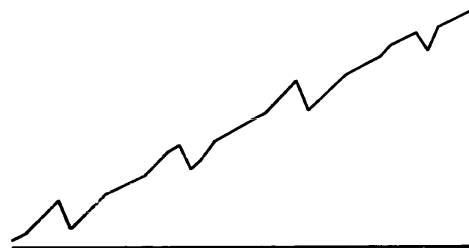
To summarize, Shellsort has proven to be quite parallelizable. However, because Shellsort has higher complexity than quicksort, the speedup of any parallel algorithm based upon Shellsort will be handicapped. For example, if for a particular problem Shellsort takes three times as long to execute as quicksort, then the speedup of any parallel Shellsort algorithm would be limited to the number of processors divided by three.



(A)



(B)



(C)

Fig. 3. Final iterations of parallel Shell-sort.

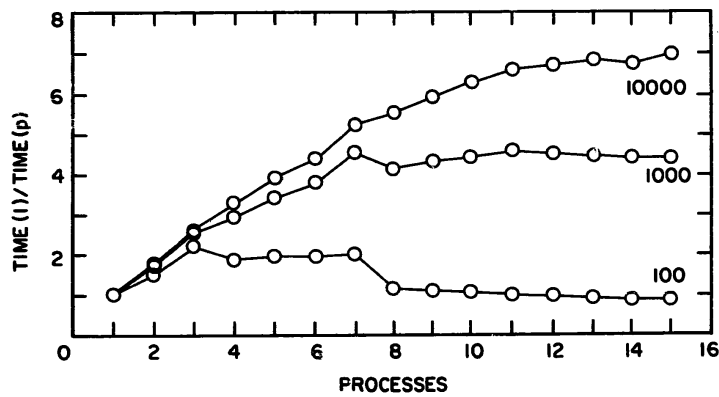


Fig. 4. Parallelizability of Shellsort.

4. Quickmerge

The parallel quicksort algorithm described in Section 2 was limited by the inefficient use of processors early in the algorithm's execution. The parallel Shellsort algorithm makes good use of its processors. The problem is Shellsort itself, which is too much slower than quicksort to be a practical basis for a parallel algorithm. We conclude that an algorithm faster than parallel quicksort will most likely be a parallelization of an $O(n \log n)$ sequential sorting algorithm.

Our new parallel algorithm, called *quickmerge*, has three phases. In the first phase each of p processes sorts a contiguous set of no more than $\lceil n/p \rceil$ keys using the fast sequential quicksort algorithm. After this phase all processes synchronize. The n keys can now be seen to form p independent sorted lists of size approximately $\lceil n/p \rceil$.

In the first sorted list of $\lceil n/p \rceil$ keys, $p-1$ evenly-spaced keys are used as dividers to partition each of the remaining sorted lists into p sublists. The second phase accomplishes the partitioning as follows. Each process i , where $1 \leq i \leq p-1$, finds, for lists 2 through p , the index of the largest key no larger than the key located at index $\lfloor in/p^2 \rfloor$ in list 1. After this phase all processes synchronize. At this point each of the sorted lists has been divided into p sorted sublists with the property that every key in every list's i th sorted sublist is greater than any key in any list's $(i-1)$ st sorted sublist, for $2 \leq i \leq p$.

In the third phase, each process i , where $1 \leq i \leq p$, performs a p -way merge of the i th sorted sublists. Note that unlike phase one, in which each process sorts a contiguous block of keys, in phase three each process merges p lists stored in p different areas. Because of the demarcations established in phase two, these merges are completely independent of each other. After this phase all processes synchronize, and the list is sorted.

Figure 5 illustrates a three-processor sort of a list of twenty-four elements using quickmerge. After the initial quicksort phase, the list consists of three sorted sublists of length 8. During the second step binary search is used to determine where the elements one-third and two-thirds of the way through the first sublist would fit in the other sublists. In the final phase each

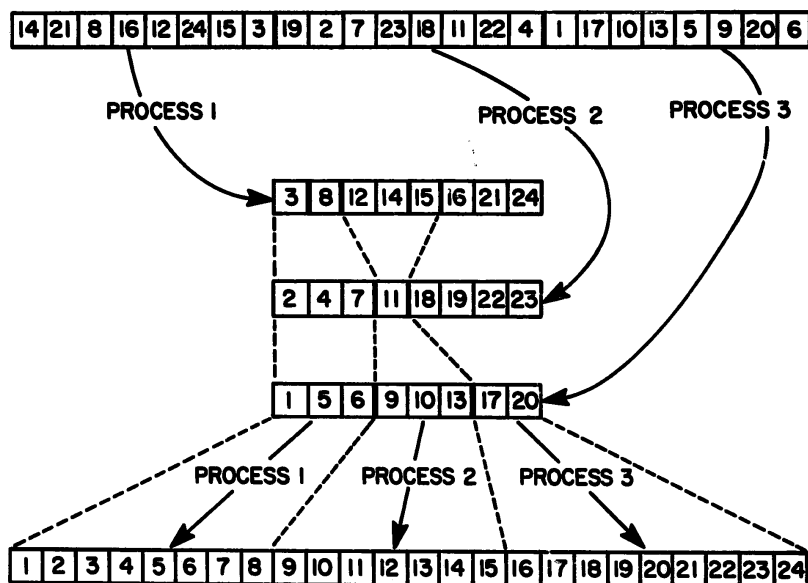


Fig. 5. Example of quickmerge.

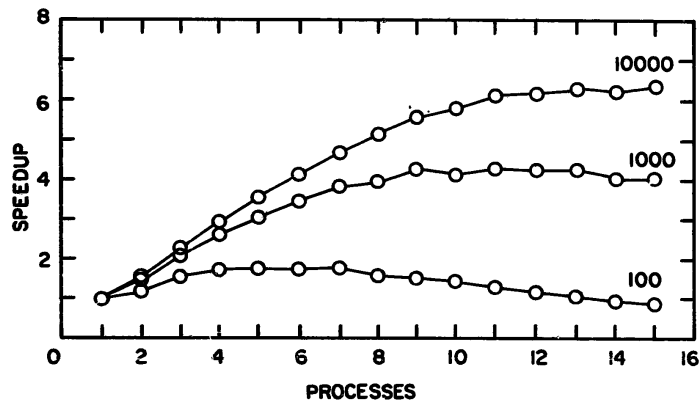


Fig. 6. Speedup of quickmerge algorithm.

processor performs a three-way merge on its own set of sorted sublists. The dotted lines connecting the sublists in the figure demarcate the keys to be merged in the final step. Note that every processor knows where to begin storing its merged list, since this is dictated by the indices of the first elements of the sublists.

Let us determine the worst-case time complexity of the quickmerge algorithm. The worst-case time complexity of sorting n keys using quicksort is $O(n^2)$. Thus, given n keys to be sorted and

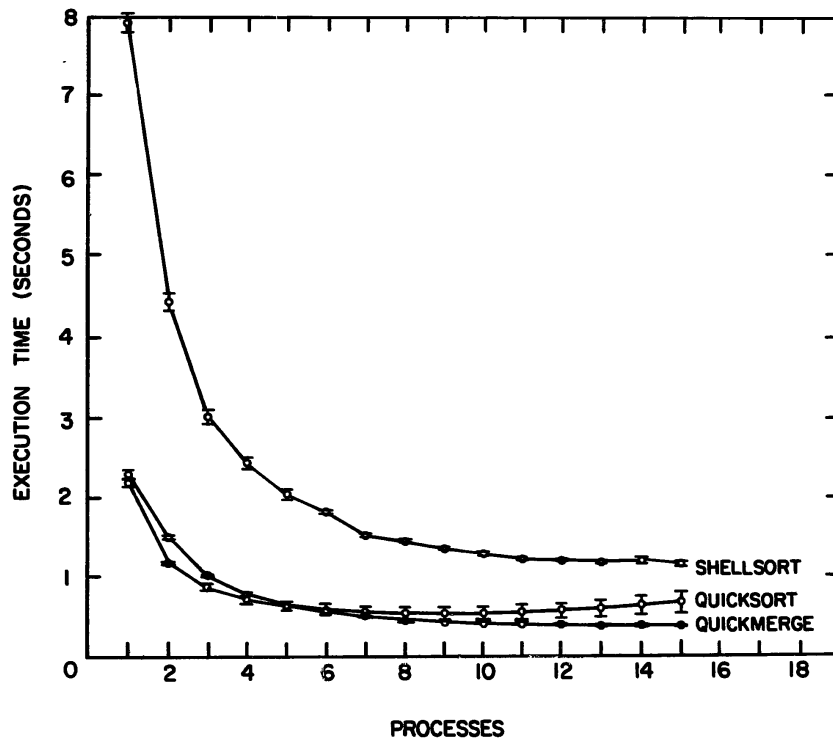


Fig. 7. Comparison of three parallel internal sorting algorithms.

p processors, the initial quicksort step has worst-case time complexity $O((n/p)^2 + p)$, where the final $O(p)$ term represents the time needed to synchronize p processors. In phase two each process must perform $p - 1$ binary searches on lists of size no greater than $\lceil n/p \rceil$; hence the worst-case time complexity of phase two is $O(p \log(n/p) + p)$. In the worst case a single processor may have to perform a merge sort on $O(n)$ keys in phase three. (For example, if the first sorted subarray contains the smallest n/p keys of the list, then after step two processor p will be given the unenviable task of merging all of the remaining keys contained in the other subarrays.) Assuming that a heap with p elements is used to perform a p -way merge in phase three, then it takes $O(p)$ time to initialize the heap and $O(\log p)$ time to perform each of $O(n)$ merge steps. Hence the complexity of phase three is $O(n \log p + p)$. Therefore, the worst-case time complexity of the quickmerge algorithm is $O((n/p)^2 + p \log(n/p) + n \log p + p)$.

We have implemented quickmerge on a single-PEM Denelcor HEP running under the HEP/UPX operating system and executed it on the same data used to test the previous algorithms. Since the single process version of this algorithm degenerates into quicksort, comparing the single processor execution time to the multiple process execution time is a fair indicator of speedup. The speedup curve is shown in Fig. 6.

Figure 7 contrasts the mean execution times of the three parallel algorithms for ten randomly permuted lists of ten thousand elements. The bars above and below each point represent one standard deviation in the execution times. Quickmerge is clearly the superior algorithm.

5. Summary

The speedup of parallel quicksort is constrained if the partitioning of a sublist is performed by a single processor. Our goal was to find algorithms more parallelizable than quicksort.

Shellsort has proven to be amenable to parallelization, even in the final few iterations when the number of independent subarrays to be sorted is less than the number of processors available. Unfortunately, Shellsort has a higher complexity than quicksort ($O(n^{3/2})$ vs. $O(n \log n)$), and its higher complexity wipes out its advantage of more inherent parallelism.

Quickmerge, a combination of quicksort and multiway merge sort, has high inherent parallelism throughout the algorithm. Our implementation of quickmerge on the Denelcor HEP exhibits better execution times than parallel quicksort.

The quickmerge algorithm as described in this paper is suitable only for a small number of processors. As the number of processors increases, so does the probability that one of the processors will have to merge significantly more than its share of keys in the third stage of the algorithm. This reduces speedup, since the final synchronization cannot complete until the last processor finishes merging. A future paper will describe modifications to the quickmerge algorithm that address this difficulty.

Acknowledgment

This work was partially supported by a University of New Hampshire Faculty Development Grant. My special thanks go to Ann Hayes, Dale Carstenson, Robert Hiromoto, and Olaf Lubeck of Los Alamos National Laboratory for providing HEP access and support. I would like to single out Robert Hiromoto, who helped me debug the parallel quicksort program during my visit to Los Alamos. The quickmerge algorithm was the fruit of a conversation I had with L. Michael Gray at the University of New Hampshire.

References

- [1] S. Baase, *Computer Algorithms: Introduction to Design and Analysis* (Addison-Wesley, Reading, MA, 1978).
- [2] J. Deminet, Experiences with multiprocessor algorithms, *IEEE Trans. Comput.* **31** (4) (1982) 278–288.
- [3] A. Gottlieb, An overview of the NYU Ultracomputer project, Ultracomputer Note #100, Ultracomputer Research Laboratory, Division of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, 1986.
- [4] R.H. Halstead, Jr., Multilisp: a language for concurrent symbolic computation, *ACM Trans. Programming Languages and Systems* **7** (4) (1985) 501–538.
- [5] C.A.R. Hoare, Quicksort, *Comput. J.* **5** (1962) 10–15.
- [6] H.F. Jordan, HEP architecture, programming and performance, in: J.S. Kowalik, ed., *Parallel MIMD Computation: HEP Supercomputer and Its Applications* (MIT Press, Cambridge, MA, 1985) 1–40.
- [7] D.E. Knuth, *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms* (Addison-Wesley, Reading, MA, 1981) 139.
- [8] M.J. Quinn, *Designing Efficient Algorithms for Parallel Computers* (McGraw-Hill, New York, 1987) Chapter 2.
- [9] L. Raskin, Performance evaluation of multiple processor systems, Ph.D. Thesis, Carnegie-Mellon University, 1978.
- [10] R. Sedgewick, Implementing quicksort programs, *Comm. ACM* **21** (10) (1978) 847–857.
- [11] Sequent, Balance 8000 System Technical Summary, Order No. MAN-0110-00, Sequent Computer Systems, Inc., Beaverton, OR, 1984.