University of Washington

Department of Computer Science and Engineering

"Technical Report TR 91-01-01"

Parallel Quicksand: Fast Sorting on the Sequent

Simon Kahan & Walter L. Ruzzo

Note: This document is *ex post facto*. The referenced TR number was assigned in 1991, but either never completed or subsequently lost (our collective memories fail completely as to which). Its closest living relative appears to be the attached abstract, derived from a LaTeX source file dated May 21, 1992, which outlines the gist of the results presumably available then. We make it available in hopes that it will appeal to CSE history buffs. – WLR 7/2013

Parallel Quicksort:
Fast Sorting on the Sequent

Simon Kahan [*]
Walter L. Ruzzo [†]

Abstract

We develop a series of quicksort algorithms for the Sequent Symmetry shared memory parallel computer. By employing a novel yet simple parallel splitting algorithm and dynamic scheduling we are able to achieve a speedup of 13 with 16 processors over the performance of sequential quicksort on one processor. This work provides experimental evidence that asynchronous algorithms' more uniform usage of shared resources may make them less susceptible to bottlenecks on real machines.

Introduction: On sequential processors, Quicksort is one of the most efficient algorithms for sorting large arrays of data [Hoa62]. Three factors are eminently responsible for its speed: locality of memory references, a tight inner loop, and, on average, a small number of comparisons. Locality ensures both that cache prefetching can be exploited and that paging is minimal. Because a high proportion of the inner loop operations are comparisons, the computational overhead per comparison is small. Even though some other sequential sorting algorithms perform hardly any more, and possibly fewer, comparisons in the worst case than Quicksort performs on the average [Weg90], Quicksort remains the algorithm of choice over a wide range of input sizes due to its efficiency on real machines.

Attempts to parallelize Quicksort are evident in the literature of both theory and practice. Some of the relevant theoretical work on parallel Quicksort is based on PRAM [FW78] models of computation in which the number of processors available is unlimited, and access to memory locations are of constant cost independent of address and of the number of processors active. Martel and Gusfield [MG89] and Chlebus and Vrťo [CV91], for example, describe formulations of Quicksort on the PRAM model sorting N distinct keys in $O(\log N)$ expected time with N processors. Other PRAM variants of Quicksort are described in Akl [Akl85] and JáJá [JáJ92]. There is nothing intrinsically wrong with such PRAM algorithms, but certain

[*]Max-Planck Institute for Computer Science, Saarbruecken, Germany. `skahan@ag1.mpi-sb.mpg.de`
[†]Department of Computer Science & Engineering, University of Washington, Seattle, WA. `ruzzo@cs.washington.edu`

pragmatic difficulties prevent their application in practice. The first difficulty is that there are no PRAMs on which to run them. One humble approximation to a PRAM is the Sequent shared memory multiprocessor: roughly 20 processors, each having a 64K cache, share a single bus accessing main memory. In principle any N processor, time T PRAM algorithm can be simulated in time $O(TN/P)$ by a $P < N$ processor machine. In practice, however, this approach seems not to lead to optimal speedup on a modestly parallel machine such as the Sequent when compared to an efficient sequential Quicksort implementation, even though the PRAM version may be asymptotically optimal. Overhead of the simulation is one reason. More seriously, considerable effort is expended by the PRAM algorithm to orchestrate the operation of the large number of available processors. This seems to have greatly complicated the algorithms, seriously undermining two of the three factors identified above as critical to Quicksort's practical efficiency on real machines. Furthermore, this effort is largely unnecessary, since as we will show, simple stratagems suffice when the number of processors is modest. Note, however, that parallelizing Quicksort is not trivial even for modest numbers of processors. In particular, the "obvious" approach of allowing one processor to sequentially perform the initial partition of the file into two halves is a poor choice, since it leaves $P-1$ of the processors idle for $O(n)$ steps, a significant waste in practice.

Access to shared memory through a single bus is an obvious bottleneck to scaling Sequent-like machines to very large numbers of processors. Increasing the bandwidth to shared memory may be achieved by replacing the bus with a communication network and main memory by a distribution of memory banks, but the result is a machine that must contend with greater latencies and more expensive coherency protocols. The hypercube is a living example of an architecture that employs such a network. Hyperquicksort [Wag91] is a version of quicksort implemented on the NCUBE, a 64 processor hypercube network. Despite its access to a vastly superior communication network, Hyperquicksort achieves in practice speedup somewhat less than what we are able to achieve on our humble Sequent. Even if the communication network is infinitely fast, expected performance of Hyperquicksort degrades significantly as the number of processors is increased, due simply to load imbalances.

Contribution: In implementing Quicksort on the Sequent, we faced the obstacles both of processor orchestration that make the PRAM quicksort algorithms inapplicable, and the potential for load imbalance as encountered by Hyperquicksort. We have confronted both obstacles with simple but apparently successful techniques. Whereas the PRAM algorithms use extra storage and/or are significantly more complex than sequential Quicksort in order to effectively apply N processors to partitioning, our algorithm retains essentially the same simple in-place inner loop, but using P processors in parallel (for modest P). Whereas Hyperquicksort suffers from poor processor utilization when splitting is uneven, our algorithm uses parallel partitioning and dynamic scheduling to obtain a roughly even load throughout the computation.

Our overall approach, of creating a large number of dynamically sized tasks during execution in order to balance load, is fundamentally different from other sophisticated implementations of parallel quicksort. Recently, Shi and Schaeffer [SS92] developed a parallel quicksort-mergesort hybrid sorting method using regular sampling to produce a partitioning of the input data into P very nearly equally sized independent subproblems. Their speedups are comparable to ours, and their algorithm is probably more scalable because it makes more effective use of caching. A drawback of their method is that the subproblems, though equally sized, may take different amounts of time to sort on an asynchronous multiprocessor, thus upsetting the even loading. Even if each sub-problem requires a similar amount of total work, the processors may run at different effective speeds due to the operating system or competing processes.

The sequel is organized as follows: We briefly present as a benchmark the sequential quicksort implementation and its performance. Then we examine the behaviour of a straightforward staticly scheduled parallel version. To reduce idle time due to processors finishing their tasks at differing rates, we implement a shared stack, or workqueue. Still, far too much time is wasted due to the sequential splitting procedure, so we implement a parallel splitting algorithm. In order to minimize synchronization and locking while preserving the fast inner loop of parallel quicksort, we employ an interlaced approximate splitting algorithm followed by a cleanup phase. The expected time complexity of the cleanup phase is small enough in comparison to the splitting time to result in a theoretically efficient sorting algorithm. While it does not achieve optimal speedup, our algorithm retains enough of the qualities of sequential quicksort to outperform a careful mergesort implementation on the same machine. We improve the speedup of our implementation still further with the addition of a simple dynamic scheduling heuristic.

Quicksorts with Sequential Splitting: As a basis for comparison, we have implemented and tested the sequential quicksort algorithm as in Sedgewick [Sed78]. When implemented on one Sequent processor, the run times in milliseconds as a function of problem size for this algorithm are listed in Table 1. Except where indicated otherwise, input to our experiments are arrays of pseudo-random integers chosen uniformly over $[0, 2^{31})$.

Table 1: Quicksort Results: Sequential code with one processor.

Size/ #Proc	1000	10000	40000	100000	1000000	2000000
1	33ms	425	1913	5233	61929	131863

One approach to introducing parallelism is modeled by the binary execution tree of quicksort. The basic idea is to utilize just one processor to perform the first partitioning, or splitting, phase of sequential quicksort. The processor continues to work on one of the resulting sub-arrays, and the other is passed on to a second processor. The two processors repeat this process – partitioning their sub-array and waking up a not yet utilized processor – until all processors have been utilized. For the remainder of the computation, each processor continues working on its subproblem just as in the sequential algorithm. We present the measured performance of this static schedule on randomly sorted lists of integers in the full paper.

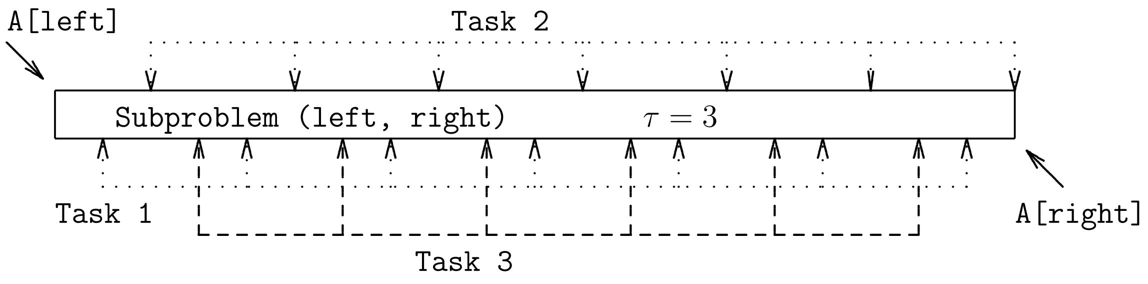
While a desirably simple algorithm, the expected performance for even a moderate number of processors, P, is far worse than the desired $O(N \log N/P)$; e.g., doubling P falls far short of halving the time. The main reason that the performance suffers is that processors are idle too much of the time. Some of the idle time is at the beginning of the computation while processors are waiting for their subarray. We call this *leading load imbalance.* In addition, there is idle time at the end of the computation, because the processors do not finish all at the same time. We call this *trailing load imbalance.*

Almost all of the trailing load imbalance in the static scheduling method is due to uneven splits; only a little is due to the disparity in numbers of elements compared that end up being swapped. Methods to reduce trailing load imbalance are discussed in the full paper.

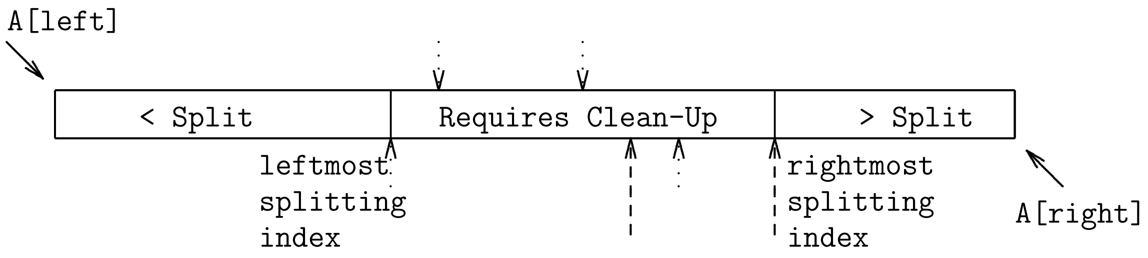
Leading load imbalance is unavoidable if we insist on a one-to-one correspondence between calls to Quicksort and processes: So long as only one processor performs the initial split, all other processors must wait for work. By allowing more than one processor to participate in splitting a subproblem in two, all processors can be utilized from the outset. In addition, by eliminating the correspondence between processors and subproblems, the task granularity can be controlled so that trailing load imbalance is further decreased.

A Quicksort with Parallel Splitting: Splitting an array consists of moving all the values in the array less than a chosen key to the lower part of the array and all values greater than the key to the upper part. Performing splitting in parallel is not as simple as doing so sequentially: the difficulty is that the final location of a general element could be anyplace in the array, and is not known a priori. Hence, the array cannot be apportioned into consecutive pieces in such a way that processors never find themselves with an element in hand that belongs in the partition assigned to some other processor. To preserve the properties that make sequential quicksort fast, an efficient parallel splitting algorithm must have a fast inner loop, perform little redundant work, and avoid synchronization and locking. The algorithm we present has all of these properties.

The basic idea behind our algorithm is to involve a number of processors in the splitting process by partitioning subproblems into a number of tasks. Each task looks very much like a subproblem in the sequential splitting approach and uses the same value as its splitting key, but the elements accessed within any task are not contiguous: instead, they are offset by as many elements as there are tasks. I.e., we divide the subarray into τ *slices,* where the i-th slice consists of those subarray elements whose indices are congruent to $i \bmod \tau$. Task i partitions the i-th slice using the usual Quicksort partitioning code.



Once all slices have been split, all elements to the left of the leftmost splitting element are smaller than the splitting element; all to the right of the rightmost are greater.



Those elements between the two extreme splitting elements' positions, while correctly partitioned within their respective slices, are not correctly partitioned globally, and need to be resplit. Since this region is small in comparison to the size of the whole subproblem (roughly of size $O(\sqrt{n} + \tau)$), the extra work is negligible. In the full paper, we describe the implementation in great detail, explaining how this partitioning can be accomplished while avoiding lock contention.

Performance of our quicksort with parallel partitioning on uniformly pseudo-random input is presented in Table 2. These times indicate a maximum speedup of almost 9 with 16 processors. Other researchers have obtained more nearly full speedup on the Sequent using other sorting algorithms. Anderson [And88] reports virtually full speedup for his implementation of mergesort on up to eight processors, but the time with a single processor to sort 40,000 elements is 7 seconds. Our quicksort sorts the same size array in under 2 seconds. So with full speedup, we can expect 8 processors to mergesort the array in no less than 874 milliseconds. With quicksort, though the speedup is less than ideal, the same array is sorted in only 227 milliseconds. Even were mergesort to achieve linear speedup on 16 processors, quicksort's time would still be faster.

In the full paper, we report even greater speedups (up to 13 with 16 processors) by employing a Polychronopoulos-style scheduler [PK87, Pol88] to further reduce trailing load imbalance.

References

[Akl85] Selim G. Akl. *Parallel Sorting Algorithms*, volume 12 of *Notes and Reports in Computer Science and Applied Mathematics.* Academic Press, 1985.

Table 2: Quicksort Results: Parallel Partitioning

Size/ #Proc	1000	10000	40000	100000	1000000	2000000
1	28	414	1918	5469	72445	154589
2	29	253	1099	3201	42892	92509
4	29	143	611	1756	23356	50454
8	29	90	332	965	12644	27327
16	29	74	227	614	8081	17554

[And88] Richard Anderson. An experimental study of parallel merge sort. Technical Report # 88-05-01, Computer Science Department, University of Washington, 1988.

[CV91] Bogdan S. Chlebus and Imrich Vrťo. Parallel quicksort. *Journal of Parallel and Distributed Computing*, 11(4):332–337, April 1991.

[FW78] Fortune and Wyllie. Parallelism in random access machines. In *Symposium on Theory of Computing*, 1978.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[JáJ92] Joseph JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.

[MG89] Charles U. Martel and Dan Gusfield. A fast parallel quicksort algorithm. *Information Processing Letters*, 30:97–102, January 1989.

[PK87] C. Polychronopoulos and D. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, December 1987.

[Pol88] C. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Norwell, Massachusetts, 1988.

[Sed78] Robert Sedgewick. Implementing quicksort programs. *Communications of the ACM*, 21(10):847–857, October 1978.

[SS92] Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, April 1992.

[Wag91] Bruce Wagar. Hyperquicksort – a fast sorting algorithm for hypercubes. In *Proceedings of the 2nd Conference on Hypercube Multiprocessors*, 1991.

[Weg90] Ingo Wegener. Bottom-up heap sort, a new variant of heap sort beating on average quick sort. In *Mathematical Foundations of Computer Science*, volume 452. Springer-Verlag, August 1990.