A full parallel Quicksort algorithm for multicore processors

Arne Maus,
Dept. of Informatics, University of Oslo
arnem@ifi.uio.no

Abstract

The problem addressed in this paper is that we want to sort an `integer array a[]` of length *n* in parallel on a multi core machine with *p* cores using Quicksort. Amdahl's law tells us that the inherent sequential part of any algorithm will in the end dominate and limit the speedup we get from parallelisation. This paper introduces ParaQuick, a full parallel quicksort algorithm for use on an ordinary shared memory multi core machine that has just a few simple statements in its sequential part. It can be seen as an improvement over traditional parallelization of the Quicksort algorithm, where one follows the sequential algorithm and substitute recursive calls with the creation of parallel threads for these calls in the top of the recursion tree. The ParaQuick algorithm, starts with *k* parallel threads, where *k* is a multiple of *p* (here $k = 8*p$) in a *k* way partition of the original array with the same pivot value, and hence we get 2*k* partitioned areas in the first pass. We then calculate where the pivot index, the division between the small and large elements if this had been ordinary sequential Quicksort partition. In full parallel we then swap all small elements to the right of this pivot index with the large elements to the left of this pivot index – these two 'displaced' sets are by definition of equal size. We can then recursively with half of the threads now do the left part, and with the other half of the threads the right part (more details and synchronization considerations in the paper). Finally, when there is only one thread left working on one such area, sequential Quicksort and Insertionsort are used, as in the traditional way of doing parallel Quicksort. In the last part of the paper, this new algorithm is empirically tested against two other algorithms and Arrays.sort from the Java library. Five different distributions of the numbers to be sorted end three different machines with $p = 2$(4 hyper threaded), 4(8) and 32(64) are tested. Finally, conclusions are presented and an explanation is given why this ParaQuick algorithm for large values of *n* and some distributions is so much faster than a traditional parallelisation.

Keywords: Quicksort, Full parallel algorithm, parallel sorting, multicore.

Introduction

The chip manufacturers cannot deliver what we really want, which is ever faster processors. The heat generated increases with the clock frequency, and will make the chip malfunction and eventually melt above 4 GHz with today's technology. Instead they now sell us multi core processors with 2-16 processor cores, but more special products with 50 to100 cores are also available [2, 22], and the race for more processing cores on a chip doesn't stop there. The Intel Xeon Phi processor with its fast, unconventional memory access and 61 cores is promising [2]. Each of these cores has the processing power of the single CPUs sold some years ago. Many of these cores, but not all, are hyperthreaded, where some of the circuitry is duplicated such that each core can switch between two threads within a few instruction cycles if the active thread is waiting for some event like access to main memory. Also, we see today servers with up to 4 such hyperthreaded multi cores processors, meaning that up to 64 threads can run in parallel. We use one of these servers in this paper. The conclusion to all this parallelism is that if we faster programs, we must make parallel algorithms for exploiting these new machines.

This paper is submitted to the NIK 2015 conference. For more information see http://www.nik.no/

The problem addressed in this paper is that we want to sort an `integer array a[]` of length *n* on a shared memory machine with *p* cores. We assume no prior knowledge of the distribution or the maximum value of the keys in `a[]`. This paper presents ParaQuick, a new full parallel version of Quicksort [1] which is demonstrated to be some 2-5 times faster than the standard Java sorting method Arrays.sort, which is an optimized sequential implementation of Quicksort with 2 partitioning elements in each step [8] and is faster than the standard way of implementing sequential Quicksort. The ParaQuick algorithm addresses the limitations posed to us by Amdahl's law [3] that basically says that any sequential part of an algorithm will sooner or later dominate the execution time of the parallel algorithm, thus limiting the speedup we can get with increased parallelism. ParaQuick does this by having only a few sequential statement before going full parallel with all *k* threads to sort `a[].` By full parallel we mean that all threads start working as soon as the ParaQuick method is called, and load balancing is done such that all threads will work at full speed with (almost) the same amount of keys to sort until all sorting is done. Waiting in ParaQuick occur at 4 synchronization points where all threads have to wait on a barrier for the other threads to get equally far in their code. None of the loops, arrays segments etc. in the threads are longer than *n/k*. In theory then, with *p* cores, we could hope for a speedup close to *p*. As suggested by one anonymous referee, if we overbook the number of threads by a factor > 1 compared with the available cores, it would give a better performance for ParaQuick. We overbook by a factor 8 here.

Parallel sorting algorithms are abundant [5,12,14,15,23,25]. As with sequential sorting, we can distinguish between comparison based methods, where the values of two (or more) keys are compared to do the sorting; and content based methods, where the value of some bits in a single key determines where it will be sorted. Most work has been done on parallel comparison based algorithms. We find first of all a set of algorithms for special purpose network machines, but also for grids of more ordinary machines [5,6,8,9,25]. The Hyperquick algorithm for a hypercube of connected machines resembles some, but not all of the ParaQuick algorithm presented in this paper[18].

In addition most textbooks on parallel computations, present algorithm following the theoretical PRAM model [5,23] of a computer. In PRAM one assumes that: i) access to any location takes the same unit of time, and ii) that we have as many cores available as one needs – often *n* cores. Both assumptions are utterly wrong. In today's multicore computer, the different between accessing data in the registers or in cache level 1 versus in main memory is a factor of 200, and between computers in a cluster there is a factor of at least 10 000. In addition, the number of cores is always limited; on a single computer it is 2-100, not *n*, and even in a cluster it is still less than 10^6. These textbooks by obvious reasons never produce actual performance figures. The most troublesome with the PRAM model is that their assumptions do not produce the most efficient algorithms, in particular the assumption of same access time to any part of data is damaging to algorithm construction because often extra or copies of data and/or extra operations are introduced to increase speed.

Also relevant to this paper is bitonic sort [8,9,13,14], a variation of the comparison based merge sort; and sample sort, a generalization of Quicksort that sort the keys into many buckets before sorting each bucket with ordinary Quicksort [11,12, 25]. A number of the algorithms seems either to be of the

Quicksort type with a relative slow start (first sequential, then two parallel, four parallel,..), or with a slow ending like merge and bitonic sort, where **parallelism** decreases as longer, but fewer sequences are merged (the last step is two-parallel in ordinary merge sort). To the best of my knowledge, the algorithm presented in this paper has not been presented in the literature.

The rest of this paper is organised as follows. In section 2 the fully parallel ParaQuick algorithm is defined. In section 3 the differences between the ParaQuick and a traditional parallel quicksort algorithm are analysed to some detail. In section 4, tests comparing ParaQuick with three other algorithms on three different machines with from 64 to 4 cores, and for 5 different distributions of numbers to be sorted, are presented. The results from these tests are analysed in section 5 and in section 6 the paper is concluded.

2. The full parallel algorithm ParaQuick defined

With *p* cores at our disposal, we for performance start *k* =8**p* threads. The first operation in ParaQuick is to divide the array `a[]` into *k* (almost) equally sized segments (Fig.1). Then the *k* threads are started. Let *m* = a.length/2, they all calculate the same pivot value with the use of Insertionsort on the same three elements `a[m-1],a[m],a[m+1];` then picking the middle one as the common pivot value. All threads then synchronize (Fig. 2) and each thread sorts its own part of length *n/k*. These segments are then partioned by the ordinary Quicksort partition algorithm (Appendix A).

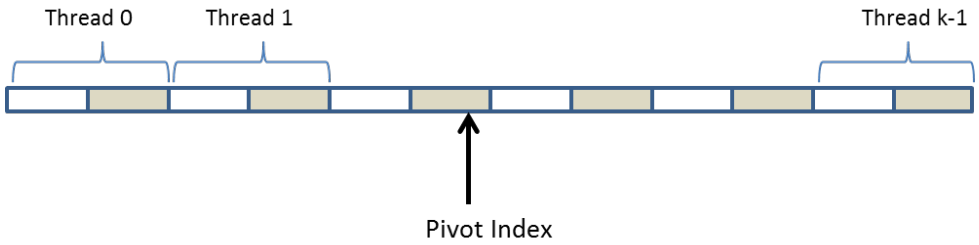

Figure 1. *Partition of a[] into k segment, with a number of 'smaller' and 'bigger' than the common Pivot value.*

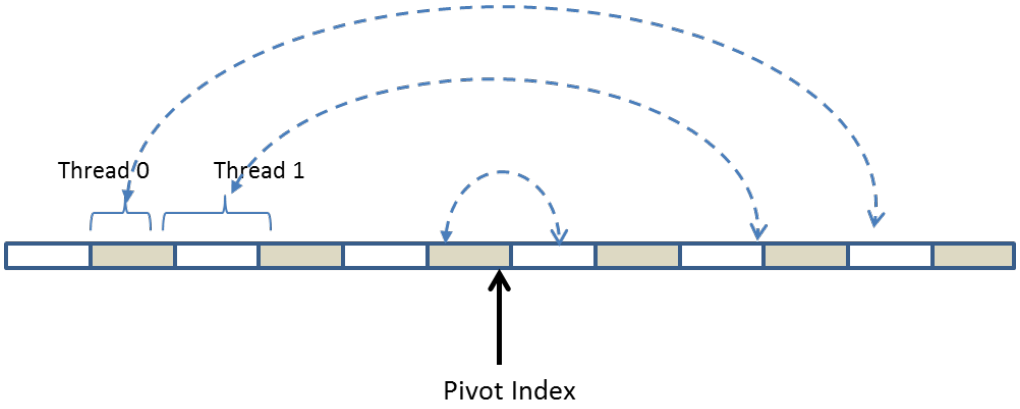

Figure 2. *Parallel swapping of all 'big' element in a[] to the right of the Pivot Index with the same number of 'small' element to the left.*

Many large elements are now to the left of the Pivot index – which can be calculated as the sum of all *k* 'small' segments. The algorithm then swaps in

parallel these displaced elements of **`a[]`** with small element to the right of this Pivot index. These two 'displaced' sets are by definition of equal size. We then synchronise on a Barrier and end up with a single partitioned array with one part that is smaller than the pivot value, and a right part that have elements that are bigger (Fig. 3) .

We see that this swapping is an overhead, extra operations that are not performed by the ordinary parallelization of Quicksort. On the other hand, we have now completely removed the top of the recursion tree together with its sequential code and low level parallelism.

An additional overhead of any parallel algorithm is the creation of threads to make this parallelism happen. The first thread started in Java costs approximately 2 to 3 milliseconds, but subsequent calls to the `new Thread` operation are much faster due to JIT (Just In Time) compilation of the running code. The results presented here are the average over many executions of sorting an array of the same length. Only when $n = 10^9$, the running time for a single run is presented.

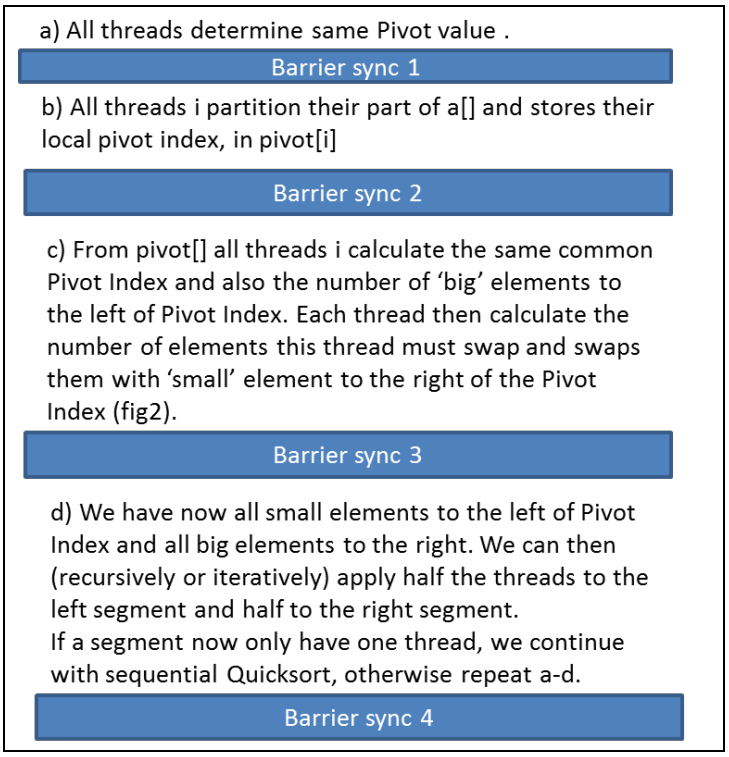

***Figure 3**. The ParaQuick algorithm and its synchronisation.*

But since a sequential algorithm can sort 50 000 numbers in 2-4 milliseconds, we cannot expect any parallel algorithm to outperform a sequential algorithm for any *n* less than 100 000.

We will in the following section empirically investigate how the ParaQuick compares with: a) The highly tuned Arrays.sort, b) A straight forward sequential Quicksort, and c) A traditionally crafted parallel Quicksort: TradParaQuick where one follows the sequential algorithm and substitute recursive calls with the creation of parallel threads for these calls in the top of the recursion tree for any section to sort longer than 50 000 elements. Data is first then sequentially split in two separate parts, each part gets a thread and we can then continue in 2-parallel. Then again each part of data is split in two, and we get a 4 parallel program, then

8-parallel. After this parallelization, which has too much sequential and too restricted parallelisation, each thread then finally uses sequential quicksort on its part of the array when its length is shorter than say 50 000.

3. The difference between the TradParaQuick and the ParaQuick algorithms analysed.

Assume that we have p cores. In TradParaQuick we first do sequentially n reads and $n/4$ swaps; at the next level it is 2-parallel. Time wise it then does $n/2$ reads and $n/8$ swaps, then $n/4$ reads and $n/16$ swaps. Asymptotically this takes the same time as doing $2n$ reads and $n/2$ swaps sequentially for the parallelisation phase.

The new ParaQuick does nothing of this, but does time wise $n/(p*2)$ extra swaps at the first level, then $n/(p*4)$ extra swaps at the next level,.., until we get down to one thread per segment. Asymptotically this is n/p extra swaps. In addition ParaQuick does three synchronisations per level and one at the end compared with one for TradParaQuick per level, and one at the end.

The reason that it pays in ParaQuick to use far more parallel threads than cores, is probably that each thread then solves a smaller sub problem that fits better, higher up in the caching hierarchy while there is almost no penalty in the rest of the algorithm against doing this.

Theoretically, we then conclude that when $p \geq 2$, the new algorithm ParaQuick should be faster than TradParaQuick.

4. The test results

Apart from Arrays.sort, we note that the other three algorithms tested use sequential partition of a segment and Insertionsort for large parts of their algorithm. To assure fair comparison, the same code fragment for these algorithms is used by all three algorithms, and is given in appendix A. All code are compiled and run with Java 8. As with Arrays.sort, all three algorithms also use Insertionsort if $n < 32$ and the two parallel ones do not start parallel sorting, but use sequential Quicksort if $n < 50\,000$.

We tested these 4 algorithms on three different machines, on new laptop with 2(4 hyperthreaded) cores, one workstation with 4(8) cores, and one server with 32(64) cores. We also tested 5 different distributions of the n randomly selected numbers to be sorted: Uniform (0:n/10), Uniform(0:n), Uniform (0:10^{30}) , Uniform (i%3) and Uniform (i%29) – the last two only having 3 and 29 possible values – and thus many duplicates.

We note that Arrays.sort is a two-way optimized Quicksort and hence faster than the simple sequential quicksort which we present as the starting point for both TradParaQuick and ParaQuick. Since we present the speedup of the three other algorithms as speedup versus Arrays.sort, this is not strictly correct. The speedup of the two parallel algorithms are actually better than given in the following figures, since it should have been calculated compared with a on an average 20% slower algorithm than Arrays.sort. However, since there is no reason to introduce a new sorting algorithm if it is not faster than the sorting algorithm in the Java library, this speedup is used.

Time is measured with the Java system call: `System.nanoTime()`. We first now present the speedup for each of the 3 machines with all the 5 distributions.

Since all execution times are presented relative to Arrays.sort for the same length of the array, it might be interesting to know what the absolute execution times are. In Table 1 the execution time for the various machines are given for sorting 100 million integers with Arrays.sort.

Time to sort 100million 32 bit integer keys from the U(0:n-1) distribution with Arrays.sort() – a sequential two way quicksort:	**Millisec.**
2(4) core Intel i7-4600U, 2,1-2,7GHz	10757
4(8) core Intel i7-870, 3Ghz	10477
32(64) core Xeon L7555 1.87GHz	14722

Table 1. *Time to sort 100 million 32 bit integers with Arrays.sort on the three machines used. We see that, the sequential performance ratios between the three machines are less than 1 to 1,5.*

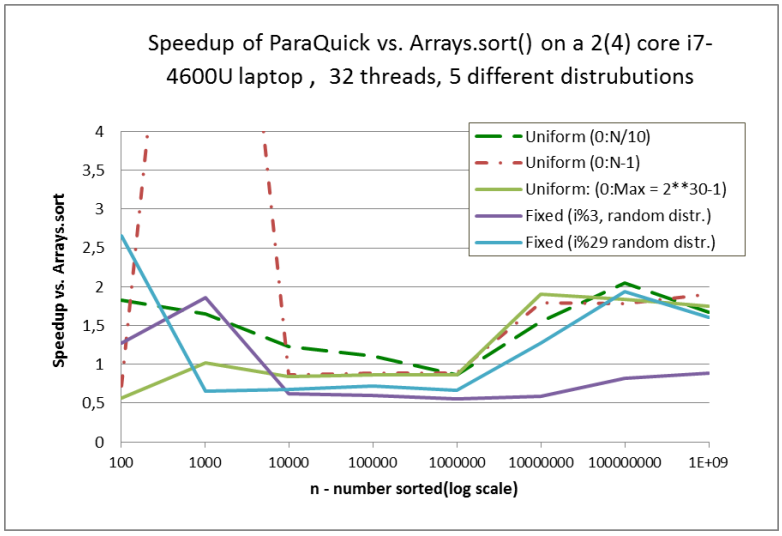

Figure 4*. Speedup on a laptop with 2(4) cores, 32 threads (max speedup for n=1000 out of graph is 11)*

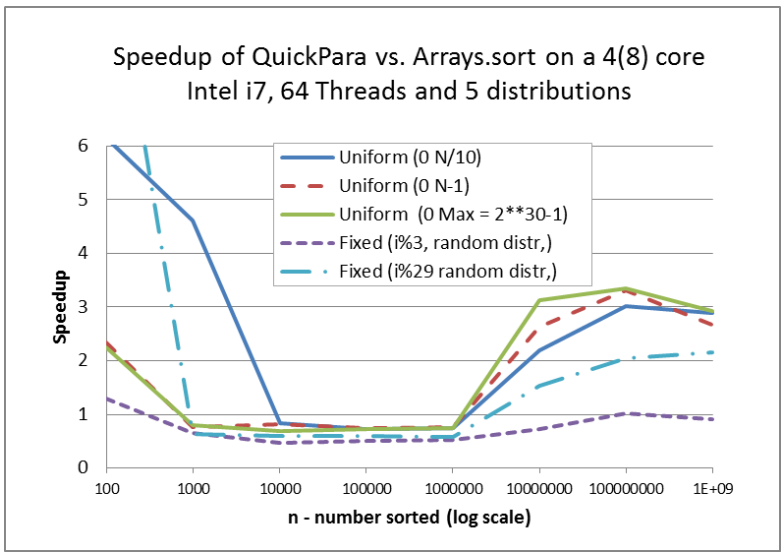

Figure 5. *Speedup for 5 distributions on a 4(8) core Workstation, 64 threads .*

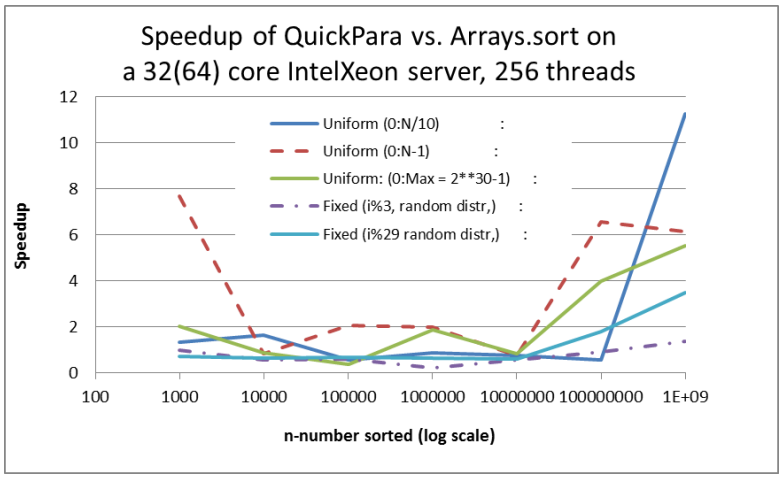

Figure 6. *Speedup for 5 distributions on a 32(64) core server, 256 threads*

To clarify points that we will make in the next section, we now present two speedup figures for all three algorithms with the two the most frequently used distributions in the sorting literature, the U(0:n-1) and U(0: 2^{30}) on the same 4(8) core workstation.

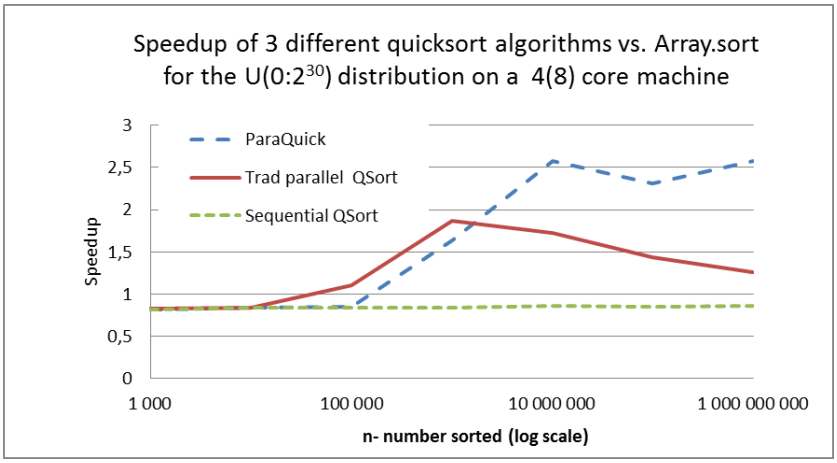

Figure 7. *Speedup for 3 algorithms with the U(0:n-1) distribution on a 4(8) core workstation.*

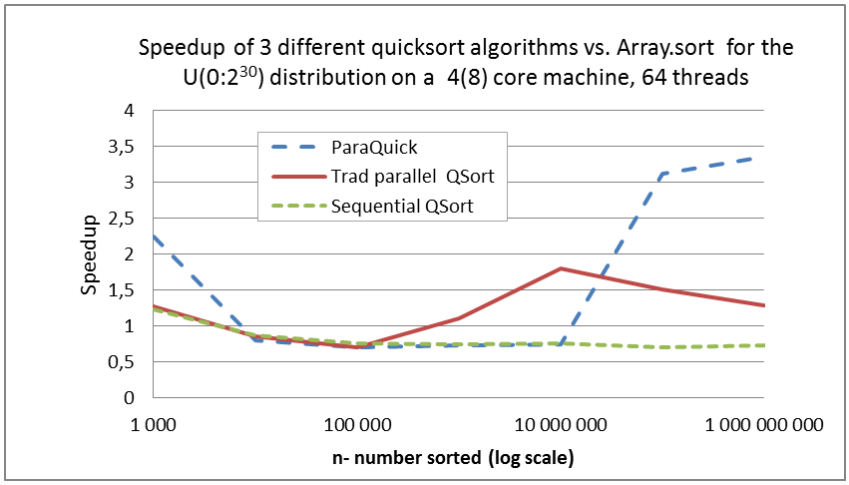

Figure 8. *Speedup for 3 algorithms with the U(0:10^{30}) distribution on a 4(8) core workstation.*

Since we have overbooked the number of threads with respect to the number of cores, this is illustrated in the last figure below where we vary the number of threads on the 4(8) core workstation for the U(0:*n*-1) distribution.

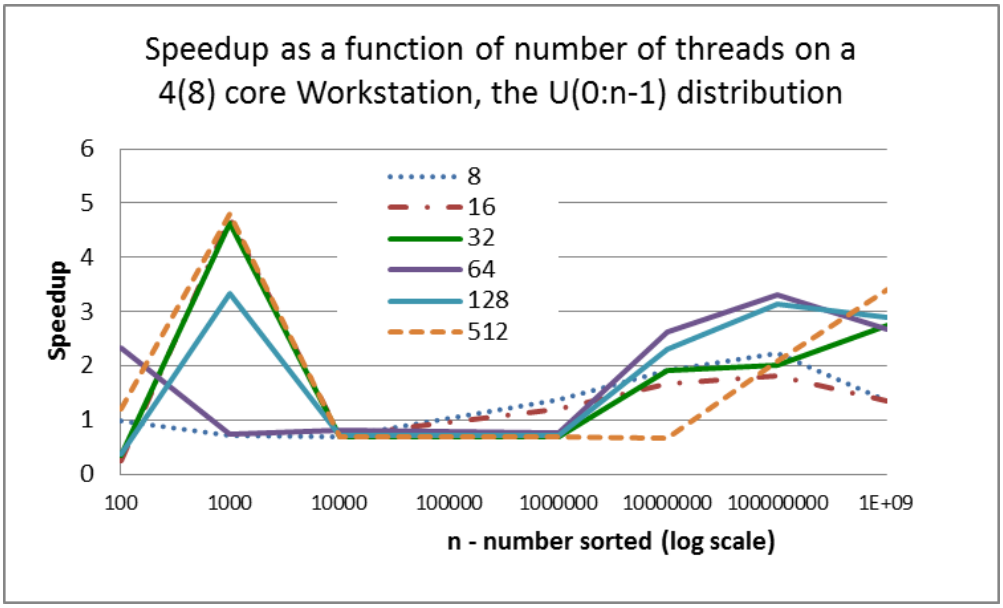

Figure 9. *Speedup as a function of number of parallel threads p used on a 4(8) core workstation for the U(0:n-1) distribution.*

5. Analysis of the test results

We see in Figures 3 through 8 that for large $n > 10^6$, ParaQuick is much faster than a traditional parallelised quicksort on three machines tested. Figures 4 to 6 demonstrate that speedup increases with *p*, the number of cores. These empirical findings support the more theoretical analysis that the overhead of ParaQuick decreases with *p*, the level of parallelism, as it also is intuitively correct. We do not swap more displaced elements as *p* increases – on the contrary, we do this in parallel so that the execution time of this extra swapping decreases as *p* increases.

We also see that overbooking the parallelism by using more threads than the number of cores as demonstrated in Figure 9 is advantageous, but of course there is a limit to useful overbooking, and $p = 8*k$ seems a reasonable choice.

The actual Java code for ParaQuick will be posted on my sorting homepage [24] for free download together with this paper.

6. Conclusion

I have presented a new algorithm ParaQuick that sorts significantly faster than the standard sequential Quicksort and a traditionally parallelised Quicksort on a shared memory computer with more than one core when $n \geq 10^6$ where sorting times matters most.

Bibliography

[1] C.A.R Hoare : *Quicksort*, *Computer Journal* vol. 5(1962), 10-15
[2] http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html
[3] http://en.wikipedia.org/wiki/Amdahl%27s_law

[4] M.J. Quinn: *Analysis and benchmarking of two parallel sorting algorithms: Hyperquick and Quickmerge*, BIT 29(1989), 239-250
[5] J. JaJa, *Introduction to Parallel Algorithms*, Addison–Wesley, Reading, MA, 1992.
[6] Frank Thomson Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann Pub, Sept. 1991, ISBN:9781558601178
[7] A. Borodin and J. E. Hopcroft, *Routing, merging, and sorting on parallel models of computation* , Journal of Computer and System Sciences, Volume 30, Issue 1, February 1985 , Pages 130-145
[8] Black, Paul E. "Multikey Quicksort". Dictionary of Algorithms and Data Structures. NIST.
[9]. K. E. Batcher, *Sorting networks and their applications*, Proc. AFIPS Conference, 1968,pp. 307–314.
[10] Y. C. Kim, M. Jeon, D. Kim, and A. Sohn, *Communication-efficient bitonic sort on a distributed memory parallel computer*, Proc. International Conference on Parallel and Distributed Systems,ICPADS'2001, Kyongju, Korea, June 26–29, 2001.
[11]. D. R. Helman, D. A. Bader, and J. JaJa, *Parallel algorithms for personalized communication and sorting with an experimental study*, Proc. ACM Symposium on Parallel Algorithms and Architectures, Padua, Italy, 1996, pp. 211–220.
[12] J. S. Huang and Y. C. Chow, *Parallel sorting and data partitioning by sampling*, Proc. the 7th Computer Software and Applications Conference, 1983, pp. 627–631.
[13] J.-D. Lee and K. E. Batcher, *Minimizing communication in the bitonic sort*, IEEE Trans. Parallel Distrib. Systems 11(5) (2000), pp.459–474.
[14] Zhaofang Wen, *Multiway Merging in Parallel*, IEEE Transactions on Parallel and Distributed Systems Volume 7, Issue 1, January 1996
[15] Amato et al : *A Comparison of Parallel Sorting Algorithms on Different Architectures*, Technical Report 98-029, Department of Computer Science, Texas A&M University, College Station, January 1996
[16] Jon Bentley, *Programming Pearls*, Second Edition, Addison-Wesley, 2000. ISBN 0-201-65788-0.
[17] Donald Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Second Edition. Addison-Wesley, 1998.
[18] P. Tsigas, Y. Zhang, *A Simple, Fast Parallel Implementation of Quicksort And Its Performance Evaluation on SUN Enterprise 10000* (2003), ELEVENTH EUROMICRO CONFERENCE ON PARALLEL, DISTRIBUTED AND NETWORK-BASED PROCESSING
[19]S. Sen and S. Chatterjee. *Towards a theory of cache-efficient algorithms.* 11th ACM Symposium of Discrete Algorithms, pages 829–838, 2000.
[20] R. Wickremesinghe, L. Arge, J. S. Chase, and J. S. Vitter. Efficient sorting using registers and caches. ACM Journal of Experimental Algorithmics, 7(9), 2002
[21] LaMarca, A.; Ladner, R. E. (1997). "The influence of caches on the performance of sorting". *Proc. 8th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA97)*: 370–379.
[22] (Tile-GX 100 core) http://www.Tilera.com
[23] R.E. Neapolitan, *Foundations of Algoriths*, Joanes&Bartlett Learning, fifth ed. 2015
[24] Arne Maus' sorting homepage at: http://arnem.at.ifi.uio.no/sorting/
[25] David R. Cheng, Alan Edelman, John R. Gilbert, and Viral Shah. *A novel parallel sorting algorithm for contemporary architectures*. Submitted to ALENEX06, 2006

Appendix A – The sequentialQsort (sub) algorithm employed in algorithms defined in this paper.

```
static void  sequentialQsort (int[] a, int left, int right) {
   // sort a[left..right],try skipping equal elements in the middle
   if (right-left < INSERT_MAX) insertSort (a,left,right); // = 32
   else {int i =left, j =right,t;
         int part = a[(left+right)/2];
         while ( i <= j) {
                   while (a[i] < part ) i++;
                   while (a[j] > part ) j--;

                   if (i <= j) {
                     t = a[j];
                     a[j]= a[i];
                     a[i]= t;
                     i++;
                     j--;
                   }
           } // end while

           while (j > left  && a[j] == part)  j--;
           while (i < right && a[i] == part)  i++;

           if  ( j-left > 0)  sequentialQsort (a,left,j);
           if  ( right-i > 0) sequentialQsort (a,i,right);
    }
} // end quickSortSek

public static void insertSort (int a[], int left, int right) {
      int i,k,t;

      for (k = left+1 ; k <= right; k++) {
            t = a[k] ;
            i = k;

            while ( a[i-1]  > t ) {
                   a[i] = a[i-1];
                   if (--i == left) break;
            }
            a[i] = t;
            } // end k
} // end insertSort
```