View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by BIBSYS: Open Journals Systems

A faster, all parallel Merge sort algorithm for multicore processors

Arne Maus (em)
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 merge sort. 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 parallelization. This paper introduces ParaMerge, a new all parallel merge sort algorithm for use on an ordinary shared memory multi core machine that has just a few statements in its sequential part. This algorithm is all parallel in the sense that by recursive decent it is two-parallel in the top node, four-parallel on the next level in the recursion, then eight-parallel until we have started one thread at a level for all the *p* cores. After the parallelization phase, each thread then uses sequential recursion merge sort with a new variant of insertion sort for sorting short subsections. ParaMerge is an improvement over traditional parallelization of the merge sort algorithm that follows the sequential algorithm and substitute recursive calls with the creation of parallel threads in the top of the recursion tree. This traditional parallel merge sort finally does a merging of the two sorted halves of *a*[] sequentially. Only at the next level does a traditional approach go two-parallel, then four parallel on the next level, and so on. After parallelization my implementation of this traditional algorithm also use the same sequential merge sort and insertion sort algorithm as the ParaMerge algorithm in each of its threads.
There are two main improvements in ParaMerge: First, the observation that merging can simultaneously be done from the start of the two sections to be merged left to right picking the smallest elements of, and at the same time from the end of the same sections from right to left picking the largest elements. The second improvement is that the contract between a node and its two sub-nodes is changed. In a traditional parallelization a node is given a section of *a*[], and sort this by merging two sorted halves it recursively receives from its own two sub nodes and returns this to its mother node. In ParaMerge the two sub nodes each receive a full sorting from its two own sub nodes of the section itself got from its mother node (so this problem is already solved). Every node has a twin node. In parallel these two twin nodes then merge their two sorted sections, one from left and the other from right as described above. The two twin sub nodes have then sorted the whole section given to their common mother node. This goes also for the top node. We have thus raised the level of parallelization by a factor of two at each level of the top of the recursion tree. The ParaMerge algorithm also contains other improvements, such as a controlled sorting back and forth between *a*[] and a scratch area *b*[] of the same size such that the sorted result always ends up in *a*[] without any copy. A special insertion sort that is central for achieving this copy-free feature. ParaMerge is compared with other published algorithms, and in only one case is a feature similar one of the features in ParaMerge found. This other algorithm is described and compared in some detail.

Finally, ParaMerge is empirically compared with three other algorithms sorting arrays of length *n* =10,20,…,50 m, and ..1000m when p=32. We then demonstrate that it is significantly faster than two other merge algorithms, the sequential and the traditional parallel algorithm.
Keywords: Merge sort, parallel algorithms, parallel sorting, multicore.

1. Introduction

The chip manufacturers have since 2004 not delivered what we really want, which simply is ever faster processors. The heat generated with an increase of the clock frequency will make the chips malfunction and eventually melt above 4 GHz with today's technology. Instead, they now sell us multi core processors with 2-16 processor cores. More special

This paper was submitted to the NIK 2018 conference. For more information see http://www.nik.no/

products with 50 to72 cores are also available [2, 21], and the race for many processing cores on a chip is also found in the Intel Xeon Phi processor with its fast, unconventional memory access and 62 to72 cores [2]. Each of these cores has the processing power of the single CPUs sold some years ago. Many of these processors, 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. To the operating system one such hyperthreaded core they act as two cores. 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.

2. The parallel algorithm ParaMerge

The problem addressed is that we want to sort an integer array a[]of length n on a shared memory machine with p cores using a parallel implementation of merge sort [1]. We assume that merge sort is a well known sequential algorithm [12, 14, 17].

There are four significant features in ParaMerge, three of which are new:

1. New: Simultaneously sorting from both ends of the two segments to be merged. This always gives a balanced merge and faster, simpler code.
2. Two-parallel in the top node, no sequential merging.
3. A new insertion sort that ensures that ParaMerge is copy-free, meaning that every move of elements from a[] to b[] or vice versa, is sorting.
4. A new contract between a mother node and its two sub nodes ensuring more parallelism.

The ParaMerge algorithm addresses to some extent 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. ParaMerge does this by having only a few sequential statement before going two-parallel in the top node in a recursion sorting a[] - partitioning a[] in a top down recursion and doing the sorting on backtrack. It is all parallel in the sense that by recursive decent it is two parallel in the top node, four parallel on the next level in the recursion, then eight parallel until we have started at least two thread for all the p cores.

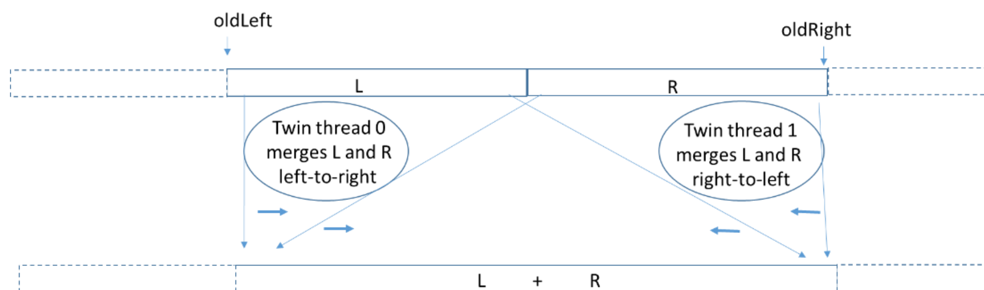

Figure 1. *Merging adjacent sorted sections L and R in parallel with two twin threads – each merging number of elements= half the sum of the length of R+L. Thread$_0$ merges L and R from* ***left*** *to right on smallest elements, while thread$_1$ merges L and R from* ***right*** *to left on largest elements.*

After parallelization with threads, each thread then uses the sequential recursion merge to sort their section. A variant of insertion sort (fig. 4) is used for sorting short subsections. Waiting in ParaMerge occurs at two synchronization points at each node in the top of the recursive tree, first when the two sub nodes have made their recursion and backtrack up to this node (fig. 2, 3). The two twin nodes at this level in the tree then sort their two adjacent sorted sections in the array (a[] or b[]) together in parallel – one from the left and one from the right. Say the combined length of these two segments is m. Then one merges from the left, and the other from the right – finding the m/2 smallest and the (m+1)/2 largest elements respectively (fig. 1). This is balanced regardless of the value of the elements to be sorted, because each thread finds a pre calculated number of elements in the two segments in a[oldLeft..oldRight] - see fig. 1. Note the difference between the code in figures 5 and 6. In the merging in fig.5 we test for having performed a fixed number of merge operations, while we in the code in fig.6 we test for end-of two sections.

The second synchronization occurs when they are finished with this and return to their mother node.

```
void rekParallel2(int [] a,int [] b, int left, int right, int level){
        // top down split , creating nodes with threads in the top
        if (right-left < BIG_LIMIT | | level > LEVEL_LIM) {
                // Sequential mergesort in each thread
            mergesortRek (a,b, left, right, level+1);
        } else {
            // start two more threads
            Thread t0,t1;
            int mid =(right+left)/2;
            CyclicBarrier barr = new CyclicBarrier(2);

            // create two threads and start
            t0 = new Thread(new Para2(a,b,left,mid,left, right, level+1,barr,0));
            t1 = new Thread(new Para2(a,b,mid+1,right,left, right, level+1,barr,1));
            t0.start(); t1.start();

            // wait for completion
             try{t0.join();t1.join();} catch (Exception e) { return ; }
         }
  } // end rekParallel2
```

Figure 2. *The indirect recursive method used to split the problem for ParaMerge. If either the problem is too small or the predetermined max level of nodes from the top or reached, a sequential merge sort is called. Otherwise two new twin nodes are created (see Fig. 3) and this method waits with join() for them to complete.*

The call to start ParaMerge is: rekParallel2(a,b,0,n-1,0); We see that the top of the recursive sorting tree is level 0, and no new thread is created for the top node; it is just the method rekParallel2 that starts two threads, one for each of the two twin sub nodes to top node.

To explain how the non-copy properties of ParaMerge is done, we note that if array a[] contains the most sorted version on backtrack when level is an even number then the whole array is sorted at level 0, the top – the last step from level 1 must then be a sort b[] to a[]. This property is secured at the bottom level at the recursion where we do insertion sort. Since the input a[] has an arbitrary length, each time we partition it in two, we will in general half of the times get two parts

where one section is one element longer than the other. We also use a formula for splitting a section in two that will always make the left part one longer if necessary. This will accumulate if we have many levels in the recursion tree, and we might end up with more levels on the left side than on the right side of the tree.

```
public void run() {
    rekParallel2(a,b, left,right,level);  // recurse further down, split problem
    // wait for my two sub twin nodes to complete
    try { barr.await();}catch( Exception e) {return;}// both t0 and t1 must have completed

    // merge together this part and twin thread ed area sorttogether
    if ( index == 0) {
            // This is left twin node
            if (evenNum(level))  mergeFromLeftHalf(a,b,left,right,oldRight,level);
            else                 mergeFromLeftHalf(b,a,left,right,oldRight,level);
      } else {
            // This is right twin node
             if (evenNum(level))  mergeFromRightHalf(a,b,left,right, oldLeft,level);
             else                 mergeFromRightHalf(b,a,left,right, oldLeft,level);
      }
 } // end run
```

Figure 3. *The run() method in the ParaMerge algorithm and its synchronizations. After the call to rekParallel2, this is the backtrack part of the nodes with new threads. We then wait on a common CyclicBarrier with its twin node to assure that the other part is sorted before we merge (index == 0 is the left twin node). The even/odd test for the parameter 'level' is to determine whether to sort a[] -> b[] or b[] -> a[].*

This is solved with a (new) formulation of Insertion sort (fig. 4) that takes two parameters – a[] and b[] and sort the section specified by its left and right parameters from a[] to b[]. The nice property of InsertSortAB is that, if given the same parameter a[] as both a[] and b[], it will just sort array a[], otherwise it will sort a[left..right] to b[left..right] without altering a[].

```
// sort a[left..right] over to b[left..right]. b might be a
 void insertABSort(int a[],int [] b,int left, int right){
        int i, t;
        b[left] = a[left];
        for (int k = left+1 ; k <= right ; k++){
              t = a[k];
              i = k-1;
             while (i >= left && b[i] > t) {
                   b[i+1] = b[i];
                    i--;
              }
              b[i+1] = t;
        } // end for k
} // end insertABSort
```

Figure 4. *A reformulation of insertion sort that either sort a[] to b[], or a[] to a[].*

This version of insertion sort has only one extra statement for each invocation (b[left] = a[left];) if it is used for sorting a[] to a[] – a small price to pay when we need this flexibility, determined at the final level to sort a[] to b[], or a[] to a[]. The call to this method is always done in the sequential part of merge sort (not shown here, but can be seen by downloading the whole code [16]).

A final note, ParaMerge is a stable sort. This is because when we sort from left-to-right, we first pick small, possibly equal elements, from the left segment. And, when sorting from the right-to-left, we first pick large, possibly equal elements, from the right segment (fig.5). Also, InsertABSort (fig. 4) is stable, hence the whole algorithm is stable.

```
void mergeFromLeftHalf(int[] fra, int [] til, int left, int right, int oldRight) {
        // merge half fra[left...] & fra[left....] from left to: til[left...]; fra->til
            int fra1 = left, t
              maxto = (oldRight+left-1)/2,
              fra2 = right+1,
              to   = left;

            while (to <= maxto ) {
                 if (fra[fra1] <= fra[fra2]) {
                      til[to++] = fra[fra1++];
                 } else { til[to++] = fra[fra2++];}
            }
    } // end mergeFromLeftHalf

    void mergeFromRightHalf(int[] fra, int [] til, int left, int right,int oldLeft) {
        // merge fra[..left-1] & fra[...right ] from right to: til[...right]; fra->til
            int fra1 = left-1,
              minto = (right + oldLeft)/2,
              fra2 = right,
              to   = right;

            while (to >= minto) {
                 if (fra[fra2] >= fra[fra1]) {
                      til[to--] = fra[fra2--];
                 } else { til[to--] = fra[fra1--];}
            }
    } // end mergeFromRightHalf
```

Figure 5. *The two methods from ParaMerge that sort half the elements from the left and half the elements from the right from two adjacent sections in a larger array fra[] to the same area in til[]. Note, only one loop with only one test.*

```
void mergeFromLeft(int[] fra, int [] til, int left, int mid, int right) {
 // merge fra[left..mid] & fra[mid+1..right] from left to: til[fra..right]; fra->til
        int fra1 = left,
          fra2 = mid+1,
           to   = left;
         while (fra1 <= mid && fra2 <= right) {
              if (fra[fra1] < fra[fra2]) {
                   til[to++] = fra[fra1++];
              } else { til[to++] = fra[fra2++];}
         }
        while (fra1 <= mid ) {til[to++] = fra[fra1++];}
        while (fra2 <= right){til[to++] = fra[fra2++];}
} // end mergeFromLeft
```

Figure 6. *Merging two segments left-to-right. Note the two tests in the central while-loop and the two extra while-loops at the end compared with the methods in Figure 5.*

3. Related work

Parallel sorting algorithms are abundant [4,5,6,9,10,11,14,16], 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 on parallelization has been done on comparison based algorithms. The home page for merge sort at Wikipedia is recommended [1]. However, the parallel merge sort algorithm presented by Arch D. Robinson in [8] is not mentioned there. This algorithm has one of the features of the ParaMerge algorithm presented in this paper.

His algorithm also does a parallel merge at all levels also at the top level. The way this algorithm does this, always from left to right, is to split the two parts to be merged. First the longest part is split in two equal parts, and the value in that middle is denoted K. In the other, smaller part to be merged, a binary search is performed to find where K would fit in there, and the value at that point is denoted K'. The second sequence is then split at that point – and we now have two sequences with small elements, one up to K and the other up to K', and the other parts are two segments with larger elements. The algorithm starts one extra thread, a copy of one of these sections is performed, and both the two sections with elements < K, and the two with larger elements can now be merged in parallel.

There are problems with the efficiency of this approach. First the binary search for K' and the creation of an extra thread and a copy. Especially the extra thread that is created and later terminated every time a merge is done, will be time consuming, also since the recursion tree is already parallelized down to any length > 2000.

Another time consuming effect is that the tests in the main loop are more complex with two extra loops at the end, compared with the very simple code in ParaMerge – compare code in fig. 6 – sorting whole sections left to right with the two methods in fig. 5 sorting half the sections from left-to-right and right-to-left. In addition, the split of the second section is will seldom be balanced. A worst case would be if all elements in the second section are greater than all elements in the first, longest section. Then this approach will be no faster than an ordinary, non parallel merge of the two original sections without a split. On the average, this approach seems not optimal.

In the following we will first analyze the differences between the ParaMerge and a traditional crafted merge sort, the TradParaMege algorithm. TradParaMege 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 say 20 000 elements or that the level is greater than some predetermined value (calculated by the number of available cores). 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 each thread then, both in ParaMerge and TradParaMerge, use the same straight forward sequential merge sort, SeqMerge on its part of the array.

Then we will empirically investigate how the ParaMerge compares with: a) SeqMerge and b) TradParaMerge. We will test these four algorithms on three different machines with 2(4 hyperthreaded) cores, 4(8) cores and 32(64) cores.

Finally, we conclude on the efficiency of the new ParaMerge algorithm.

4. The TradParaMerge and the ParaMerge algorithms analyzed.

Assume that we have p cores. In TradParaMerge we first do sequentially n reads and n *writes*; at the next level it is 2-parallel. Time wise it then does $n/2$ reads and $n/2$ writes, then $n/4$ reads and $n/4$ writes until there is no more parallelism. It then goes on doing *2p* sequential merge sort, time wise they take $\frac{n}{p} * (n * \frac{logn}{p})$ time. In table 1 this is summarized for p =2, 4,..32.

The new ParaMerge does time wise n/2 read and writes at level 1, then n/4 read and n/4 writes on the next level,… But on the level where there is not parallelism left, each last node only issues one single call to SeqMerge to sort the section it has been given. The reason for this is to keep the contract that the next node in the tree shall return the whole section sorted that was given to its mother node.

The formulas in table1 summarizes the expected running times for the TradParaMerge and ParaMerge algorithms for p=2, 4,..32.

# cores	TradParaMerge	ParaMerge
p = 2	$n+\frac{n}{2}+\frac{2n}{4}\log\left(\frac{n}{4}\right)$ $=\frac{n}{2}*(3+logn-\log 4)$ $=\frac{n}{2}(logn+1)$	$\frac{n}{2}+\frac{n}{2}\log\left(\frac{n}{2}\right)$ $=\frac{n}{2}*(1+logn-\log 2)$ $=\frac{n}{2}(logn)$
p = 4	$n+\frac{n}{2}+\frac{n}{4}+\frac{2n}{8}\log\left(\frac{n}{8}\right)$ $=7\frac{n}{4}+\frac{n}{4}*(logn-\log 8)$ $=\frac{n}{4}*(logn+4)$	$\frac{n}{2}+\frac{n}{4}+\frac{n}{4}\log\left(\frac{n}{4}\right)$ $=\frac{n}{4}*(3+logn-\log 4)$ $=\frac{n}{4}(logn+1)$
p = 8	$n+\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\frac{2n}{16}\log\left(\frac{n}{16}\right)$ $=15\frac{n}{8}+\frac{n}{8}*(logn-\log 16)$ $=\frac{n}{8}*(logn+11)$	$\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\frac{n}{8}\log\left(\frac{n}{8}\right)$ $=\frac{n}{8}*(7+logn-3)$ $=\frac{n}{8}(logn+4)$
p = 16	$n+\frac{n}{2}+..+\frac{2n}{32}\log\left(\frac{n}{32}\right)$ $=31\frac{n}{16}+\frac{n}{16}*(logn-\log 32)$ $=\frac{n}{16}*(logn+26)$	$\frac{n}{2}+\frac{n}{4}+..+\frac{n}{16}\log\left(\frac{n}{16}\right)$ $=\frac{n}{16}*(15+logn-4)$ $=\frac{n}{16}(logn+11)$
p = 32	$n+\frac{n}{2}+..+\frac{2n}{64}\log\left(\frac{n}{64}\right)$ $=63\frac{n}{64}+\frac{n}{64}*(logn-\log 64)$ $=\frac{n}{32}*(logn+57)$	$\frac{n}{2}+\frac{n}{4}+..+\frac{n}{32}\log\left(\frac{n}{8}\right)$ $=\frac{n}{32}*(31+logn-log32)$ $=\frac{n}{32}(logn+26)$

Table 1. *The expected execution times for the two parallel merge sort algorithms with p= 2, 4,.., 32 as a function of n = the length of a[].*

In the calculations in table1, the extra time used by an extra synchronization in ParaMerge and the use of insert sort for the shortest subsections are not considered. Both algorithms use the same SeqMerge algorithm as a sub algorithm which is assumed to have a running time of $\boldsymbol{n * logn}$ when sorting an array of length n.

From the results in table 1 we can calculate expected speedup as:

$$S = \frac{n\ logn}{parallel\ execution}$$

# cores	TradParaMerge	ParaMerge
p=2	$\frac{n \log n}{n/2(\log n + 1)} = 2\left[\frac{logn}{logn + 1}\right]$	$\frac{n \log n}{n/2(\log n)} = 2$
p=4	$\frac{n \log n}{n/4(\log n + 4)} = 4\left[\frac{logn}{logn + 4}\right]$	$\frac{n \log n}{n/4(\log n + 1)} = 4\left[\frac{logn}{logn + 1}\right]$
p=8	$\frac{n \log n}{n/8(\log n + 11)} = 8\left[\frac{logn}{logn + 11}\right]$	$\frac{n \log n}{n/8(\log n + 4)} = 8\left[\frac{logn}{logn + 4}\right]$
p=16	$\frac{n \log n}{n/16(\log n + 26)} = 16\left[\frac{logn}{logn + 26}\right]$	$\frac{n \log n}{n/16(\log n + 11)} = 16\left[\frac{logn}{logn + 11}\right]$
p=32	$\frac{n \log n}{n/32(\log n + 57)} = 32\left[\frac{logn}{logn + 57}\right]$	$\frac{n \log n}{n/32(\log n + 26)} = 32\left[\frac{logn}{logn + 26}\right]$

Table 2. The expected speedup of the TradParaMerge and ParaMerge algorithms with p=2,4,..,32 as a function of the length of the sorted array.

We note that ParaMerge seems to be a faster algorithm than TradParaMerge. However, the difference is not large when p =2, but increases as p ≥ 4. We also note that ParaMerge does not completely escapes Amdahl's law. It is under-parallelized by being first 2, then 4 parallel and it's not up to a full parallel algorithm before, for p =32, the 5th level in the tree. The penalty it pays for this, for p=32 : $logn/(logn + 26)$, we might call the scale down factor. TradParaMerge has a larger scale down factor for the same n and p.

A note on hyperthreaded cores. When a processor says that it has p cores in Java, it counts both real cores and the hyperthreaded ones. It is debatable how much additional speed one gets from hyper-threading, numbers from 0% to 30% increase [27] has been reported. In the above tables for simplicity, p is assumed to be all real, not hyperthreaded, cores.

5. The test results

As with Arrays.sort, all three algorithms also use Insertion sort if $n < 45$ and the two parallel ones do not start parallel sorting, but use the same sequential merge sort, MergeSeq, if $n < 20\ 000$ or the level in the tree has exceeded some predefined limit, which is a function of the numbers of cores reported by the Java

```
LEVEL_LIM = (int) Math.log(Runtime.getRuntime().availableProcessors()*2)+ 1;
```

The reason for the *2 factor is that roughly half the threads we have started are waiting higher up in the tree, and we only want to start enough threads to fill the last level of nodes with active threads.

We tested these 4 algorithms using Java8 on three different machines, on laptop with 2(4 hyperthreaded) cores, one workstation with 4(8) cores, and one server with 32(64) cores, and the numbers to be sorted was the U(n), a uniform distribution 0:n.

Time is measured with the Java system call: `System.nanoTime()`. We here present the speedup for the three tested algorithms on the three machines.

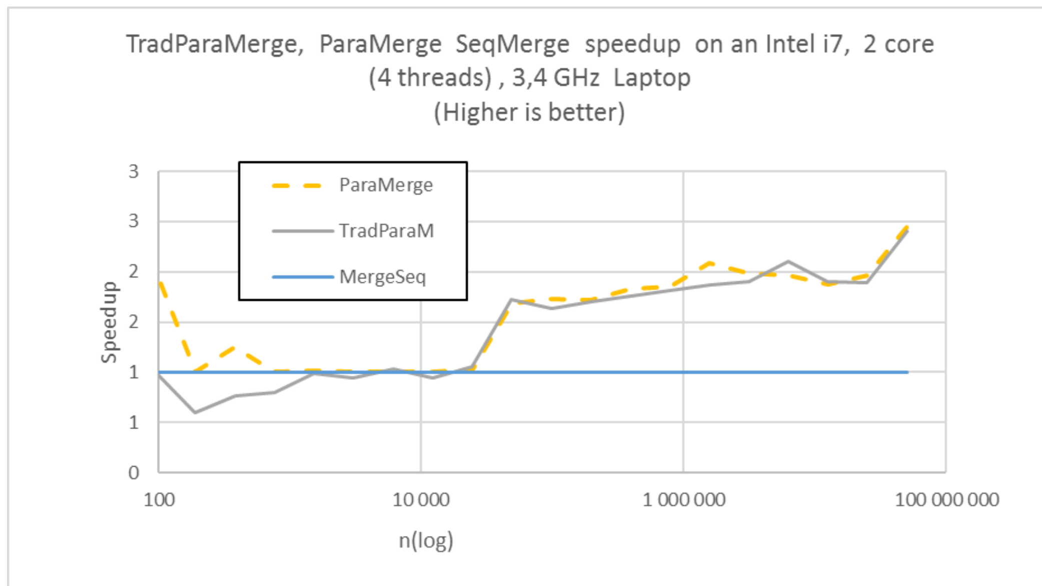

Figure 8. Speedup of the two sequential, and the two parallel and merge algorithms on a 2(4) core laptop with Intel i7-4600 CPU, 2.1 GHz.

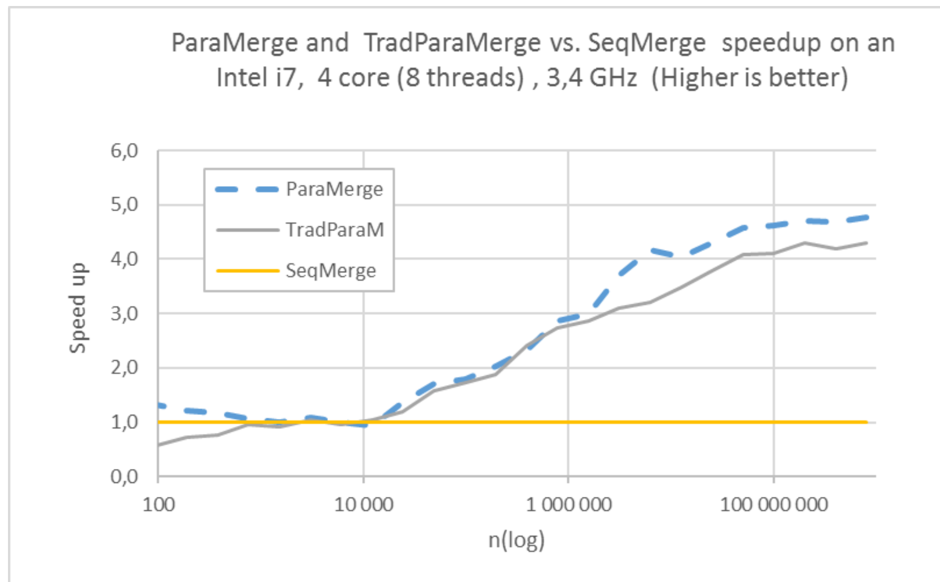

Figure 9. *Speedup of the three algorithms the sequential Arrays.sort, and the two parallel TradParaMerge and ParaMerge on a 4(8) Inteli7-7600 CPU, 2.1 GHz desktop.*

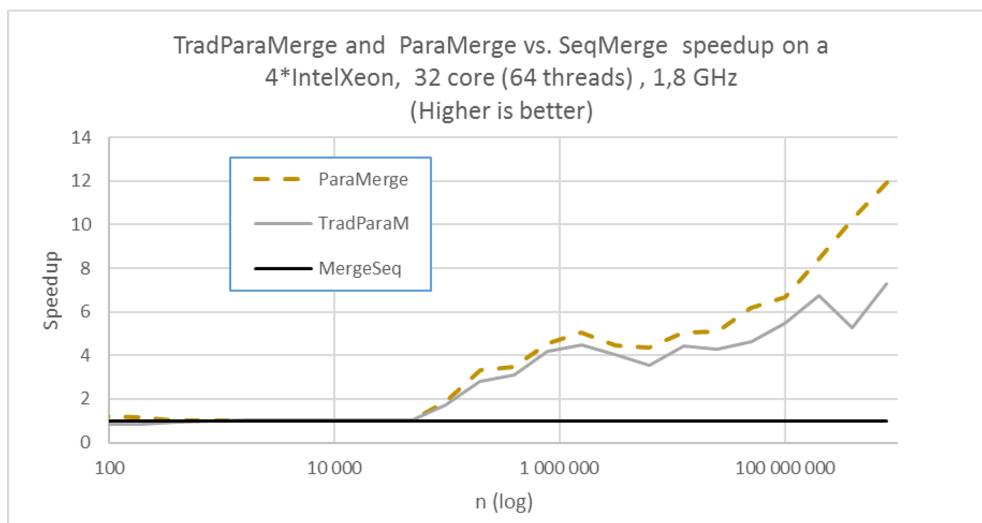

Figure 10. *Speedup of the three algorithms the sequential Arrays.sort, and the two parallel TradParaMerge and ParaMerge on a 32(64) core, 4* Intel Xeon L7555 1.87GHz server.*

6. Analysis of the test results

From the theoretical analysis, we see four interesting predictions:

a) The speedup differences between TradParaMerge and ParaMerge increase with p, the number of cores.

b) The ParaMerge is faster than TradParaMerge, at least when p >2.

c) The relative $Speedup/p$ will decrease with a larger p. (when n = 2^{26} i.e. 64mill. the scale down factor with p =32, is 0.5 for ParaMerge and 0.31 for TradMergePara. I.e., we will at best get a speedup of 16 for ParaMerge and 10 for TradParaMerge)

d) The value of n where speedup > 1 first occurs will increase with p.

The empirical tests confirm all these four patterns, but they need a further comment. For p = 2 and p=4 we get a better speedup than 2 and 4 for ParaMerge. The main reason for this is that the additional hyperthreaded cores have a positive effect on speedup. The below than expected speedup for p=32 can, apart from the scale down factor discussed above, be explained by congestion on the data channels of this 32(64) core server trying to feed data from main memory to 32 parallel threads.

The actual Java code for ParaMerge will be posted on my sorting homepage [23] for free download together with this paper.

7. Acknowledgement

I thank Stein Krogdahl for many useful comments to an earlier version of this paper.

8. Conclusion

I have presented a new parallel algorithm ParaMerge that sorts significantly faster than the standard sequential Mergesort and a traditionally parallelized merge sort on a shared memory computer with more than two cores and f $n \geq 20\ 000$, where sorting times matters most.

Bibliography

[1] https://en.wikipedia.org/wiki/Merge_sort
[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] Arch D. Robinson :"Parallel Merge Sort", Ch.13 in: M. McCool, Arch D. Robinson, J Reinders*: Structured Parallel Programming*", ISBN-10: 0124159931, Morgan Kaufmann, 2012
[9] 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.
[10] Zhaofang Wen, *Multiway Merging in Parallel,* IEEE Transactions on Parallel and Distributed Systems Volume 7, Issue 1, January 1996
[11] 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
[12] Donald Knuth, *The Art of Computer Programming, Volume 3: Section 5.2.4:Sorting and Searching,* Second Edition. Addison-Wesley, 1998. ISBN:0-201-89685-0
[13] (Mellanox multi core CPUs), https://www.mellanox.com/
[14] R.E. Neapolitan, *Foundations of Algoriths*, Joanes&Bartlett Learning, fifth ed. 2015
[15] Arne Maus' sorting homepage at: http://arnem.at.ifi.uio.no/sorting/
[16] David R. Cheng, Alan Edelman, John R. Gilbert, and Viral Shah. *A novel parallel sorting algorithm for contemporary architectures*. Submitted to ALENEX06, 2006
[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms, Third Edition. ISBN: 9780262033848, MIT Press, 2009
[18] https://en.wikipedia.org/wiki/Hyper-threading