A full parallel radix sorting 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 on a multi core machine with k cores. 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 of that algorithm. This paper introduces PARL, a parallel left radix sorting algorithm for use on ordinary shared memory multi core machines, that has just one simple statement in its sequential part. It can be seen as a major rework of the Partitioned Parallel Radix Sort (PPR) that was developed for use on a network of communicating machines with separate memories. The PARL algorithm, which was developed independently of the PPR algorithm, has in principle some of the same phases as PPR, but also many significant differences as described in this paper. On a 32 core server, a speedup of 5-12 times is achieved compared with the same sequential ARL algorithm when sorting more than 100 000 numbers and half that speedup on a 4 core PC work station and on two dual core laptops. Since the original sequential ARL algorithm in addition is 3-5 times faster than the standard Java Arrays.sort algorithm, this parallelisation translates to a significant speedup of approx. 10 to 30 for ordinary user programs sorting larger arrays. The reason that we don't get better results, i.e. a speedup equal to the number of cores when the number of cores exceeds 2, is chiefly explained by a limited memory bandwidth. This thread pool implementation of PARL is also user friendly in that the user calling this PARL algorithm does not have to deal with creating threads themselves; to sort their data, they just create a sorting object and make a call to a thread safe method in that object.

Keywords: Parallel sorting, multicore, Left Radix, PPR, ARL, PARL, Quicksort.

Introduction

The chip manufacturers can not deliver what we really want, which is ever faster processors. The heat generated increases with the square of the clock frequency, and will make the chip malfunction and eventually melt above 4 GHz with today's technology. Instead they sell us multi core processors with now 2-12 processor cores, but working prototypes of 50 to100 cores have been demonstrated [21, 22], and the race for more processing cores on a chip doesn't stop there. Each of these cores has the processing power of the single CPUs sold some 10 years ago. Some of these cores are also hyperthreaded, where some of the circuitry is duplicated such that each core can run two threads in parallel with no or little interference. 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.

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 k cores [17]. We assume no prior knowledge of the distribution or the maximum value of the keys in a[]. This paper presents a new full parallel version PARL of the ARL [2] (Adaptive

This paper was presented at the NIK 2011 conference. For more information see http://www.nik.no/

Left Radix) sequential sorting algorithm that has been demonstrated to be some 3-5 times faster than the standard Java sorting method Arrays.sort, which is an implementation of Quicksort [3]. This parallel algorithm was developed independently, but was found in some ways to be a further development of the Partitioned Parallel Radix Sort (PPR) [1] algorithm that was developed for a network of communicating machines with separate memories. The main differences between these two algorithms are described in the section 4. The PARL algorithm addresses the limitations posed to us by Amdahl's law [4] 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. PARL does this by having only one 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 PARL 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 PARL might occur at 5 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 etc. in the threads are longer than n/k. In theory then, with k cores, we should get a speedup close to k.

Parallel sorting algorithms are abundant [15, 26]. 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 [1,5,6,8,9, 27]. In the early days, content based algorithms were published that for sorting n numbers required n^2, n or some fixed fraction of n processors to sort the n keys. Most relevant to this paper are bucket sort [7] and bitonic sort [8,9,13,14], a variation of the comparison based merge sort; sample sort, a generalization of Quicksort that sort the keys into many buckets before sorting each bucket with ordinary Quicksort [11,12, 26]; and of course the contents based parallel radix sort [1]. 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). When radix sort is parallelised, it is almost without exception using the Right Radix sorting (sorting the rightmost, least significant digit first) algorithm. PPR, starts with Left Radix sorting in its first phase, but then continues with a one-bit Right Radix, while PARL uses only Left Radix sorting.

The rest of this paper is organised as follows. In section 2 it is explained how we use a thread pool to implement the fully parallel PARL algorithm, and in section 3 the pseudo code for the PARL algorithm is given. In section 4 the differences between the PPR and PARL algorithms are explained. In section 5, tests comparing PARL with three other algorithms on four different machines with from 32 to 2 cores, are presented. The results from these tests then are analysed in section 6. In section 7 the paper is concluded.

2. Reducing overhead and ensuring thread safety

A part of the sequential overhead of any parallel algorithm, is the creation of threads to make this parallelism happen. To start threads in Java costs approximately 2 to 3 milliseconds, and adds substantiality to the sequential overhead for small problems. To implement PARL efficiently, a special design pattern is used for generating a thread pool of k threads (an old idea), the same number of threads as processor cores. This is done when creating an object of the class ParaSort. The class ParaSort also contains the sorting method. In its constructor, this object starts the k threads in an ever lasting loop controlled by two CyclicBarriers. The threads then wait for the next call to the synchronized sorting method that implements PARL. Each call to the sorting method in one such sorting object will then just let all the threads do one more iteration in their loop, sort the array, and finally, let the calling main thread be released when finished.

Because of the sorting object, PARL sorting is thread safe in the sense that a sorting object will only sort one array at a time; two threads can, each with their own sorting object, sort two different arrays in parallel - or use one and the same sorting object and sort two different arrays in sequence with no interference or data race occurring. However, if two or more threads try to sort the *same* array by the use of two or more sorting objects simultaneously, the result is ill defined (but that would also be a logical programming error).

3. The ARL and the PARL algorithms

The sequential ARL sorting algorithm was first published in 2002 and is a left radix recursive decent sorting algorithm sorting first on the leftmost, most significant digit. The algorithm will only be briefly described here for two reasons: It is well documented in [2], but more importantly, it is not necessary to have a detailed understanding of the sequential ARL algorithm to understand PARL. It is only needed to have a understanding of how any radix sorting algorithm sorts an array a[] on one digit and the supporting integer array border [] used to determine where to put the keys with different values in the sorting digit.

A integer left radix sorting algorithm, like ARL, should first find the maximum value in the part of an array a[] it is called to sort. This defines how many bits in the keys we have to sort and this number of bits is then divided into one or more digits (for simplicity here assumed to be of the same length, m bits wide, say 6-11 bits). As an example, if we want to sort 1 million uniformly drawn integers less than 1 million, we then have to sort the keys in a[] on their 21 lowest bits 20-0; the upper bits 21-31 will all be zero. The algorithm then does two scans of all the keys in a[]. First it counts how many keys there are with each possible value: 0,1,..2^m-1 of the sorting digit. This counting is done in array border[0.. 2^m-1]. Then these values are added such that border[j] will point to where in a[] to store the next key with value ‘j’ in the sorting digit. In the next scan of the keys, each key is stored in its new place by its value ‘j’ on the sorting digit as specified by the appropriate value in border[j], which is afterwards increased by 1.

In ordinary Radix (Right Radix) this correct placement is done by copying between two arrays of length *n* according the value of the keys in the sorting digit and the values in border[]. In ARL, however, the sorting on a sorting digit it is done by moving the keys in their permutation cycles. We then pick up the first

unmoved key a[h], and by its value *j* of the sorting digit, border[j] will then directly point to where a[h] should be placed, but before we place it in a[border[j]], we pick up the number there. Say it has value *k* on the sorting key, but before we place it the number at a[border[k]] we pick up the number there,.. and so on until we find a number that should be placed where we found the first key in this permutation cycle. We then store the last element, and have completed one permutation cycle, which may have any length between 1 and n, and all keys moved by this cycle has now been sorted on this digit. We then find the next unmoved key in a[], moves it in a new permutation cycle until eventually all keys have been moved in a permutation cycle. This is sorting which does not need an extra array of length n as right radix does. Sorting by permutation cycles is also as fast as right radix, but is not stable sorting – meaning that equal valued elements on input might be interchanged after ARL sorting.

What is also important for understanding PARL, is that after having moved all keys in a[] to their proper places on the current sorting digit, all values in border[] points one place below in a[] where all keys with that value have been stored (Figure 1).

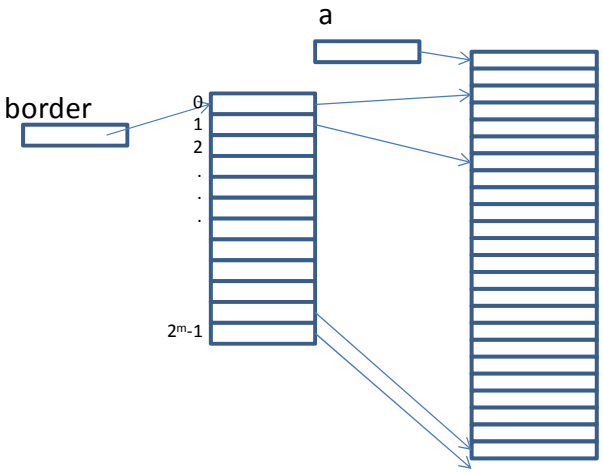

Figure 1. *Explaining the use of array border in a radix algorithm when sorting on a digit with m bits. The illustration is after sorting on that digit, and it is here assumed that the whole array a[] is sorted; a section of a[] can also, as with ARL, be sorted this way. We see that there are two keys in (this sorted part of) a[] with the value 0 on that digit, 4 keys with value 1,…,and 1 key with value 2^m-1.*

We can now give the pseudo code for the sorting done in thread i by the call to threadSortARL(a,i). The arrays a[], localMax[], allBorders [][] and bucketSize[] are all variables in the sorting object of class ParaSort, in addition to the Cyclic Barrier **`compute`** that is used to synchronize the threads when they need to read results produced by other threads from shared memory. The only sequential statement in PARL is assigning the parameter value 'a' to a local array pointer. After that statement, the threads in the thread pool are started. The threads then perform all the sorting.

This is then the pseudo code for thread i:

```
------  1 Wait on the start of the threads in the pool ----------------

localMax[i] =   max value in a[i*n/numThreads,...,(i+1)*n/numThreads-1]
```

```
------  2. Wait on the ‘compute’ Barrier'-----------------

 globalMax = max( localMax[0...numThreads-1]) //find global max value in a[]

 <all threads now calculate the same number of bits to sort, and the
 same size for the first sorting digit>

   // sequential ARL then sort  the i’th part of a[] on the first digit
   // return border-array

 allBorders[i]= oneDigitARL(a, i*n/numThreads, (i+1)*n/numThreads-1,globalMax);

 ------  3. Wait on  ‘compute Barrier'-----------------

//  put the keys on the values of first digit (0,1,.., 2^m-1) in
//  numThreads buckets. Thread 'i' will sort  the i'th such bucket
// on the rest of the digits (2,3..)  with sequential ARL for each value in bucket

 <loop through allBorders[][] and count mi= number of keys in bucket 'i'>
 bucketSize [i] = mi;

 <make a local array b = new int[mi] >

  <loop though all values in bucket ‘i’ > {
     <copy the relevant keys in a[] to b[] in for that value>
     <sort these keys on digits 2 , 3,.. using recursive, sequential ARL>
   }
 ------ 4. Wait on  ‘compute Barrier'-----------------

    <sum values in bucketSize[] to find place in a[] for the now
     totally sorted set of keys in b[]>

    <copy all keys in b[] back to relevant continuous segment in a[ ]

 ------ 5. Finished – wait for next sorting in the thread-pool ----------
```

Program 1. *Pseudo code for a thread in PARL. It first sorts on the first digit the i’th part of a[]. Then it copies a bucket of all keys with the i’th set of values on this first digit from all the different parts of a[] to a local array b[]. And for each value in this set, it sorts b[] on the rest of the digits with the sequential ARL algorithm; and finally, copies b[] back to a[].*

Some comments to the pseudo code. The most important thing to notice is that first thread i sorts on the first digit, the i’th first part of a[]. Then it takes responsibility for a *different* set of keys. We then consider the different values of the first sorting digit, and by reading the ‘border’ arrays for all the threads, copies from a[] to a local array b[] all keys with the i’th set of such values in that digit: $[(i-1)*2^m : i*2^m -1]$, after first having counted how many that is. In other words, thread i now have the set of the i’th largest numbers in a[] sorted on the first digit. Thread i then fully sorts b[] by using the sequential ARL *for each value* on the first digit in b[]. This can be done because the keys in b[] are copied into b[] sorted on the first digit – i.e. first all keys starting with $(i-1)*2^m$ are sorted, then all starting with $(i-1)*2^m$ n+1,... .As its last operation, thread i copies its b[] back to its proper place in a[].

Note also that when the ‘border’ arrays from each thread are made available to the other threads, only pointers to them are copied to the two-

dimensional array ‘allBorders[][]‘ because all the border arrays are in shared memory. Java’s way of having pointers to arrays showed itself to be very efficient here.

The pseudo code doesn’t show that any very short sub segment during sorting, say less than 32 keys, are sorted with Insertion Sort, because that is faster. Also arrays, say with fewer than 10 000 - 100 000 keys are sorted with the ordinary sequential ARL algorithm when the PARL method is called (10 000 is used for the laptops, 100 000 for the PC and server). The reason for this last decision is again that it is faster. Even using the described thread pool, there is still an overhead in the order of one millisecond when starting and stopping all threads, and the sequential ARL algorithm sorts 100 000 integers in 2,3 milliseconds on the PC. This overhead is, however, constant and hence soon a very small part of the total execution time for larger arrays.

If we compare the PARL with sequential ARL, we see that in PARL all keys in a[] are copied twice, first to the b[] arrays in each thread and then back to a[]; where as in sequential ARL, there is no such copying.

4. The differences between the PARL and PRR algorithms

Since PARL have many features in common with PPR, the main differences between the two algorithms are worth noticing:

- PPR works on memory distributed among a set of p machines communicating with MPI [25], while PARL uses a shared memory model and synchronizes access to that memory using CyclicBarriers in Java when needed.
- Two of the phases in PPR do an all-to-all communication between the p machines, an $O(p^2)$ operation. That is not needed in PARL, where the threads read the same information from shared memory in parallel.
- The initial step in PPR, the distribution of 1/p of the keys to the each processor, is not needed in PARL where the threads only work different sections on the array a[] in shared memory.
- The step of finding the *g* most significant bits in the array to be sorted, is different and a clear improvement in PARL compared with PPR. PARL does a parallelized computation of the maximum value of all keys to be sorted and uses the next 10 bits as the most significant bits. PPR, however, uses first the 8 leftmost bits in the word (bits:31-25 for 32 bit integers) and tests if that gives a good spread of values. If we only sort small numbers, these bits are all 0, and PPR then iteratively increases the number of bits it tests on by 2 each time until it gets enough bins with different valued keys.
- As mentioned above, PPR uses a mix of Left and one-bit right Radix. After having distributed the keys to each machine using left radix, it then turn to one-bit right radix sorting for the rest of the sorting. which is special in that it does not first count how many there are of each value, but only place each element in either the queue with value ‘1’ or the queue with value ‘0’ on the current digit. To sort on *k* bits in this way, each key is moved k times using *k* sorting digits, each one bit long. PARL exclusively uses Left Radix sorting, which is generally

faster and does inline sorting in the array [2] – no need for extra arrays, or queues as one-bit right Radix does.

- While the communication is explicit coded in PPR, the general tread pool pattern in PARL hides much of this from the programmer of a parallel algorithm, and hides it completely from the ordinary user .
- Sorting with PARL is almost as user friendly (ParaSort p = new ParaSort(); p.pARL(a);) as calling the ordinary Arrays.sort() method in Java. The interface to the user is not explained in PPR.
- The current implementation of PARL does not do stable sorting; meaning that same valued keys on input are not always sorted in the same order on output. It is unclear from the paper defining PPR[1] and how it does 'bucket Sort' in phase 1 of its algorithm, whether or not PPR sorting is stable. However, PARL could be made stable by parallelizing the stable version of the sequential ARL algorithm [23], but that would be a somewhat slower algorithm.

There may be other minor differences, but the above differences seem enough to consider PARL as a separate algorithm.

5. The test results

The PARL algorithm was compiled using Java 1.6 on a Unix HP based server and using Java 1.7 on one Windows 7 based PC workstation and two Windows 7 based laptops. To test the PARL algorithm, the same data was also sorted with the standard Java library Arrays.sort algorithm, which is a Quicksort [5] implementation. Also a parallel version of Quicksort was tested, and the ARL sequential algorithm [] was run on the same data to compare performance.

Just a few notes on the parallel Quicksort used for comparison. It uses the sequential Quicksort code proposed by Nico Lamuto [16] as a starting point. It first calculates how many times it can split the array until the number of cores equals the number of segments – i.e. for how many levels in the top of the recursion tree we should start new threads. Deeper that level, the sequential Quicksort is used, and very short segments are also here sorted with Insertion sort. We also use the sequential Quicksort for shorter arrays than 100 000 keys for the same reason than for ARL, it is faster. No thread pool of tasks is used and no optimality is claimed for this implementation, it is just included for comparison. But as Quicksort goes, the first split of a[] in two is done by sequential code, and on the next recursion level only two cores can be used. If we have up to 64 cores it takes some time before parallel Quicksort uses all the available cores.

The *n* integer keys to be sorted are drawn from a uniform 0:n-1 distribution, and for the server, up to 300 million keys are first sorted, for the PC Workstation 250 million keys and for the Laptops a maximum of 100 million keys are sorted. From the maximum number, then two arrays of maximum/2 length are sorted, then four times arrays of length maximum/4,.., until a number less than 100 for all machines is reached. The reason for this multiple sorting of smaller arrays, is to get more accurate time estimates. Time is measured with the Java system call: System.nanoTime().

Since, except for Figure 1, 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. The three fastest machines have hyperthreaded cores.

Time to sort 100million 32 bit integer keys with Arrays.sort() – sequential quicksort:	Millisec.
2 (2)Dual core 1.6 GHz	37493
2(4) core Intel i5 560 , 2,55 GHz	14143
4(8) core Itel i7, 3Ghz	10624
32(64) core XenonL7555 1.87GHz	18975

Table 1. *Time to sort 100 million 32 bit integers with Arrays.sort on the four machines used. We see that, except for the smallest laptop, the sequential performance ratios between the three fastest machines are less than 1 to 2.*

It is not easy to produce graphs that demonstrate how the four algorithms perform relative to one another when n varies from 100 to 250 000, and sorting times from 1 milliseconds to almost one minute. The data in figure 1 this is the *same data* as in figure 3 but without logarithmic scales on the y- and x-axis. In this authors opinion, this graph totally hides the performance for sorting short arrays less than 1 million keys, and is not used in the rest of this paper.

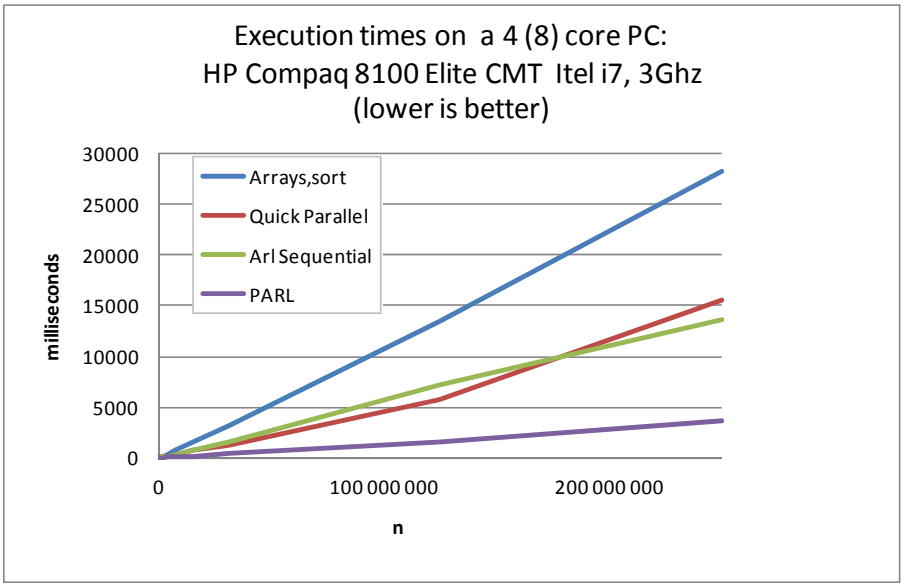

Figure 1. *The absolute execution times of four sorting algorithms when sorting an array of length n of uniformly distributed (0:n-1) keys on a 4 core (hyperthreaded) PC workstation. For n= 250mill, 125mill,...(divided by 2 until n <100). This is the same data as in Figure 3.*

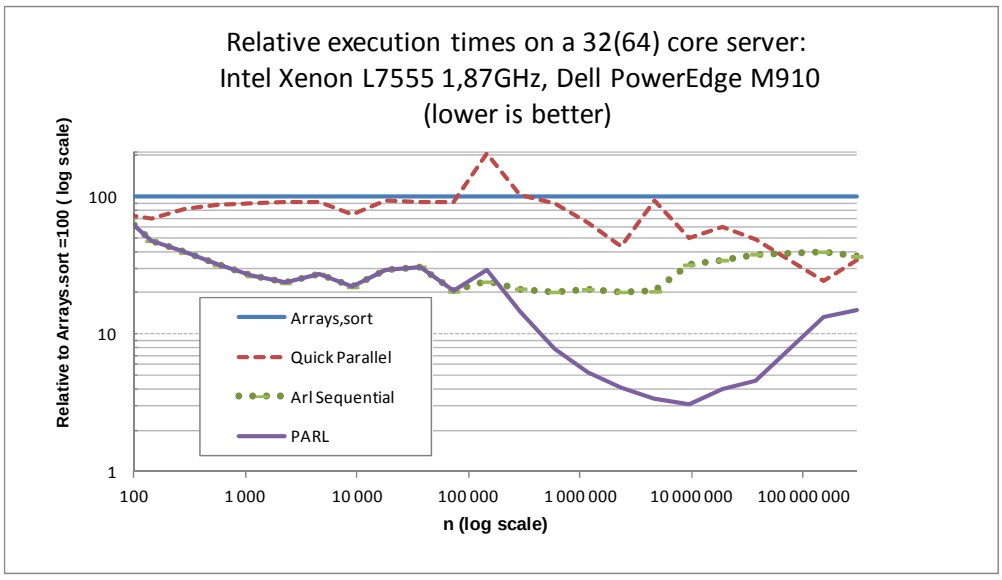

Figure 2. *The relative performance of three sorting algorithms when sorting an array of length n of uniformly numbers compared with sequential Quicksort(Arrays.sort) on a 32 core (64 hyperthreaded) server. For n= 300mill, 150mill,.....,143, 71(divided by 2 until n <100).*

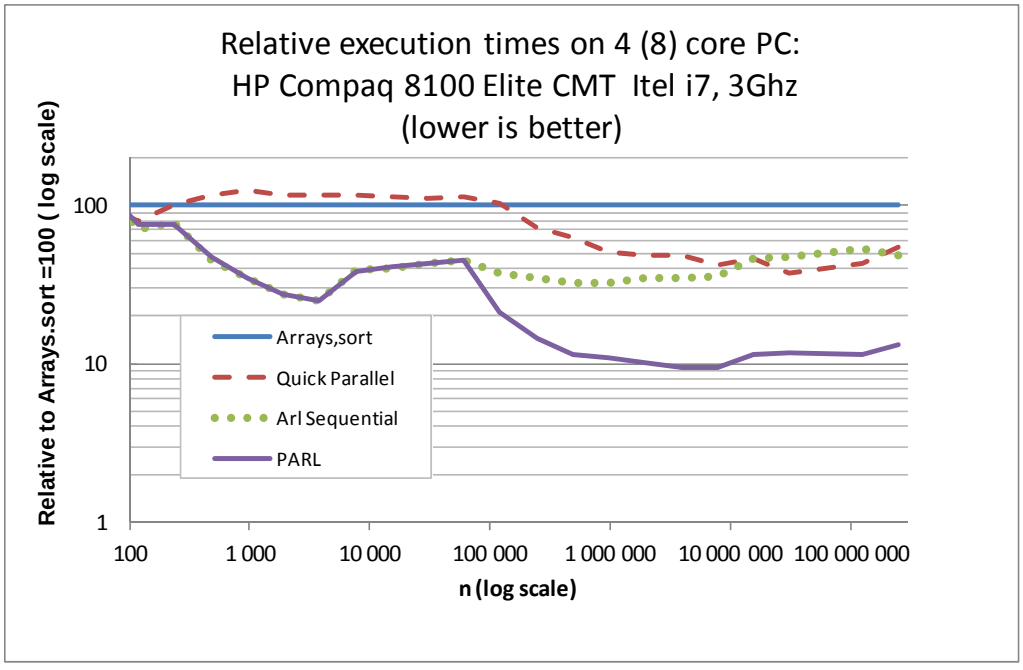

Figure 3. *The relative performance of three sorting algorithms when sorting an array of length n of uniformly integers compared with sequential Quicksort (Arrays.sort) on a 4 core (hyperthreaded) PC workstation. For n= 250mill, 125mill,... (divided by 2 until n <100).*

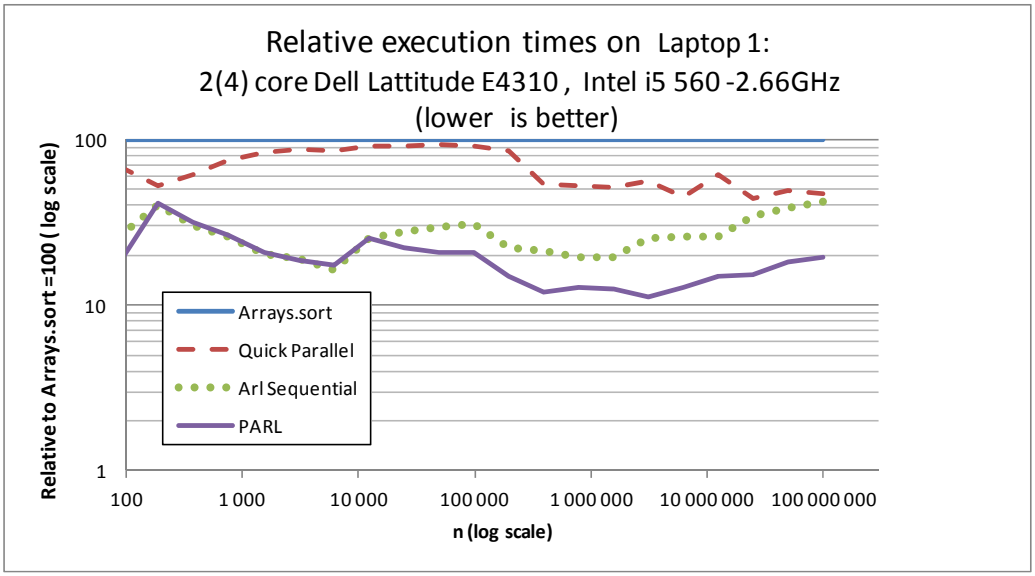

Figure 4. *The relative performance of three sorting algorithms when sorting an array of length n of uniformly distributed integers compared with sequential Quicksort(Arrays.sort) on a high end laptop, with a hyperthreaded dual core For n= 100mill, 50mill,...(divided by 2 until n <100).*

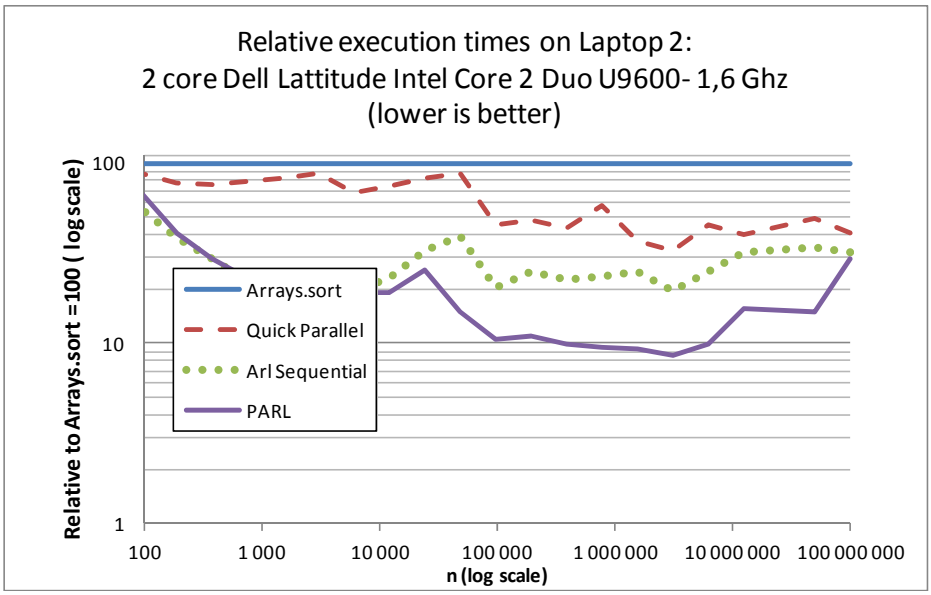

Figure 5. *The relative performance of three sorting algorithms when sorting an array of length n of uniformly distributed integers compared with sequential Quicksort(Arrays.sort) on a low end laptop, with a non-hyperthreaded dual core For n= 100mill, 50mill,...(divided by 2 until n <100).*

6. Analysis of the test results

We see that for n > 100 000 on the server and the PC, and for n > 10 000 on the laptops, when PARL is different from sequential ARL, it is roughly 5-10 times faster than the sequential ARL on a 32(64) core server, 2-4 times faster on a 4(8) core PC, and twice as fast on the two laptops with 2 and 2(4) cores. On the slowest laptop, it is sometimes more than 2 times as fast (super scaling) which can be described as the effect of caching. We have then halved the size of our problem, more of the data are in the caches, and since they are much faster than

main memory, we can expect such an effect [18,19,20]. Also the effect of halving and parallelizing a problem which is O (n*log n), should also in theory be more than twice as fast since: n*log n > 2*(n/2*log n/2).

However, the reason we in machines with more than 2 parallel threads don't get execution times at least inversely proportional to the number of cores, must be explained by the throughput of the memory channels connecting the processing cores to the main memory. Sorting is a very data intensive application. The memory channels simply can't cope with all the reading and writing of data from 64, 8 or 4 parallel threads – queues will form. This problem has also been recognized by industry that a new organization of access to main memory is needed.

The parallel Quicksort is somewhat faster than Arrays.sort even when no parallelism is employed. That must be the effect of the very tight loop in the Lamuto formulation of Quicksort. By and large, for larger values of n, this parallel Quicksort is twice as fast as Arrays.sort.

The sequential ARL is again 3-5 times faster than Arrays.sort on all machines, but we see that on the server, sequential ARL is up to 10 times slower than PARL.

We can, however, be rather satisfied with these results – that PARL sort 10 to 30 times faster for sufficiently large values of n than the standard Java sort method, is a rather good result. Also, since the user does not have to be aware of threads, this is a user friendly solution. They only create a sorting object and a pointer to that (p = new ParaSort()), and every time they want to sort an array a[], they just call the sorting method through that pointer (p.pARL(a)).

7. Conclusion

I have presented a new algorithm PARL that sorts significantly faster than the standard sequential Quicksort on a shared memory computer with more than one core. Sorting times are significantly lower, a speedup in the order of 20 to 30 when sorting larger arrays, where sorting times matters most.

I have also used a parallel design pattern for keeping a pool of threads that reduces overhead when doing repeated calls to the parallel algorithm. The threads involved are hidden from the user who doesn't need to know about threads at all. This thread safe PARL algorithm implementation will be made public on my sorting home page[24].

8. Acknowledgements

I will like to thank Stein Krogdahl for commenting on an earlier version of this paper and the referees for suggesting quite a number of improvements to the submitted paper.

Bibliography

[1] Shin-Jae Lee, Minsoo Jeon, Dongseung Kim, Andrew Sohn, Partitioned Parallel Radix Sort, Journal of Parallel and Distributed Computing, Volume 62, Issue 4, April 2002, Pages 656-668, ISSN 0743-7315, DOI: 10.1006/jpdc.2001.1808.

[2] Arne Maus. *ARL, a faster in-place, cache friendly sorting algorithm*. in NIK'2002, Norwegian Informatics Conf, Kongsberg, Norway, 2002 (ISBN 82-91116-45-8)

[3] C.A.R Hoare : *Quicksort*, *Computer Journal* vol. 5(1962), 10-15

[4] http://en.wikipedia.org/wiki/Amdahl%27s_law
[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] Bogdan S. Chlebus, *A parallel bucket sort*, Information Processing Letters,vol.27, Issue 2, 29 February 1988, Pages 57-61 doi:10.1016/0020-0190(88)90092-0
[8] 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
[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] Arne Maus and Stein Gjessing: *A Model for the Effect of Caching on Algorithmic Efficiency in Radix based Sorting*, The Second International Conference on Software Engineering Advances, ICSEA 25.Aug. 2007, France
[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] www.intel.com/go/terascale
[22] (Tile-GX 100 core) http://www.Tilera.com
[23] Arne Maus, *Making a fast unstable sorting algorithm stable*, in NIK'2006, Norwegian Informatics Conf, Molde, Norway, 2006 (ISBN 978-82-519-2186-2)
[24] Arne Maus' sorting homepage: http://www.heim.ifi.uio.no/~arnem/sorting/
[25] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Technical Report, University of Tennessee, Knoxville, TN, June 1995.
[26]David R. Cheng, Alan Edelman, John R. Gilbert, and Viral Shah. *A novel parallel sorting algorithm for contemporary architectures*. Submitted to ALENEX06, 2006
[27] Lasse Natvig. *Logarithmic Time Cost Optimal Parallel Sorting is Not Yet Fast in Practice!* in Practice!", Proc Supercomputing 90, IEEE, p486-494, 1990