Parallelization of Sorting Algorithms

Narek, Abroyan
National Polytechnic University of Armenia
Yerevan, Armenia
e-mail: narek.abroyan@gmail.com

Robert, Hakobyan
National Polytechnic University of Armenia
Yerevan, Armenia
e-mail: rob.hakobyan@gmail.com

ABSTRACT

In this work we supply several approaches of parallelization of sorting algorithms. There are a lot of sorting algorithms that differ from each other by their effectiveness, resource usage etc. We show three different types of parallelization for sorting algorithms - parallelization through algorithm modification, multithreading, OpenMP. In this work as an example we represent and develop parallel versions of bubble sort and merge sort algorithms. Bubble sort is considered as one of the most simple, but inefficient sorting algorithms. There are many other more preferable sorting algorithms that are widely used. This work shows that parallelization of bubble sort makes it quite effective and can be compared with other more effective and widely used sequential sorting algorithms. Merge sort is considered to be a more complex and effective algorithm, but we represent a more effective version of it. We also do compare the speedup for each parallel version and find out a more efficient parallel algorithm. In this work we use C++ programming language (with the new C++11, C++14 standards) for developing our algorithms.

Keywords

Sorting algorithms, bubble sort, merge sort, multithreading, OpenMP, speedup

1. INTRODUCTION

Many computer scientists consider sorting to be the most fundamental problem in the study of algorithms. Many engineering issues come to the fore when implementing sorting algorithms. The fastest sorting program for a particular situation may depend on many factors, such as the memory hierarchy (caches and virtual memory) of the host computer, and the software environment. Many of these issues are best dealt with at the algorithmic level, rather than by "tweaking" the code [1].

In computer science there are many sorting algorithms such as bubble sort, insertion sort, merge sort, quick sort, etc. They differ in their functionality, performance, resource usage. In this work as an example we will focus on bubble sort and merge sort algorithms. Bubble sort is considered as the oldest sorting algorithm. It is popular, but inefficient sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order [1]. Bubble sort has $O(n^2)$ average complexity. The best case of bubble sort is $O(n)$, it is when the input list is already sorted. Merge sort is a divide and conquer algorithm. It divides unsorted list into n list, that each of them contains only one element. After that it repeatedly merges subsists and form the final list. Merge sort has $O(n \cdot log(n))$ average complexity. It's complexity for the best case is also $O(n \cdot log(n))$.

In this work our goal is to find parallelization methods for sorting algorithms and demonstrate them on bubble sort and merge sort algorithms which will make them work faster than the original versions. At the end we will compare all the developed parallel versions.

2. THE NEED OF PARALLELIZATION

From 1986 to 2002 the performance of microprocessors increased, on average, 50% per year. Since 2002, however, single-processor performance improvement has slowed to about 20% per year. This difference is dramatic: at 50% per year, performance will increase by almost a factor of 60 in 10 years, while at 20%, it will only increase by about a factor of 6. By 2005, most of the major manufacturers of microprocessors had decided rather than trying to continue to develop ever-faster monolithic processors, manufacturers started putting multiple complete processors on a single integrated circuit. This change has a very important consequence for software developers: simply adding more processors will not magically improve the performance of the vast majority of serial programs. Such programs are unaware of the existence of multiple processors, and the performance of such a program on a system with multiple processors will be effectively the same as its performance on a single processor of the multiprocessor system [2]. So increasing a program's performance there is a need to parallelize that program, especially the algorithms that are used in that program.

One of the most important criteria of a program's parallelization is the speedup. It shows how many times the parallel program works faster than the sequential one, when both programs are solving the same problem. If T_S is the execution time of the sequential program for our problem and T_P is the execution time of the parallel program used to solve the same problem, then the speedup formula is

$$S = \frac{T_S}{T_P} \tag{1}$$

If we run our program on p cores, then the best case would be when

$$T_S = \mathrm{T_P} \cdot p \tag{2}$$

This is called a linear speedup. But According to Gene Amdahl's law even in ideal parallel programs it's hard to get such a result, because in every program there is some α fraction that cannot be parallelized. The parallel execution time and the speedup will be

$$T_P = T_S \cdot \alpha + T_S \cdot (1-\alpha)/p$$

$$S = \frac{T_S}{T_P} = \frac{p}{\alpha \cdot (p-1)+1} \tag{3}$$

And when $p \to \infty$

$$\lim_{p \to \infty} S = \frac{1}{\alpha} \tag{4}$$

This means that a program's speedup cannot exceed the number in (4). In this work we will compute the speedup by (1) for bubble sort's each parallel version and compare them.

3. PARALLELIZATION THROUGH ALGORITHM MODIFICATION

In this version of parallelization we concentrate on algorithm modification. This method is very individual for every sorting algorithm. We will develop parallel versions of bubble sort and merge sort independently.

3.1. Parallelization of bubble sort

In that case two instructions can be executed in parallel if they satisfy Bernstein's conditions. Let P_i and P_j be two program segments. For P_i, let I_i be all of the input variables and O_i the output variables, and likewise for P_j. P_i and P_j are independent if they satisfy

- $I_j \bigcap O_i = \varnothing$,
- $I_i \bigcap O_j = \varnothing$,
- $O_i \bigcap O_j = \varnothing$, [3]

One of the parallelization ways of sorting algorithms can be modification of the algorithm in a way of those operations in loops and frequently executed instructions satisfy Bernstein's conditions. In this case parallelization can be done through a pipeline mechanism. Every iteration can be performed independently from the previous one.

In case of bubble sort algorithm, each iteration's operations depend on the previous iteration's result, so there is a need to make some change in order to make possible the instructions' parallel execution. For that purpose we can use Odd-Even transposition method and divide our input list into two imaginary lists – odds and evens. On one pass through the list, we can compare an odd index and the right adjacent even index element; in the succeeding phase, it compares an even index and the right adjacent odd index element. The odd and even phases are repeated until no exchanges of data are required. This will allow instructions of each loop to be independent. Schematically this is represented in Fig. 1.

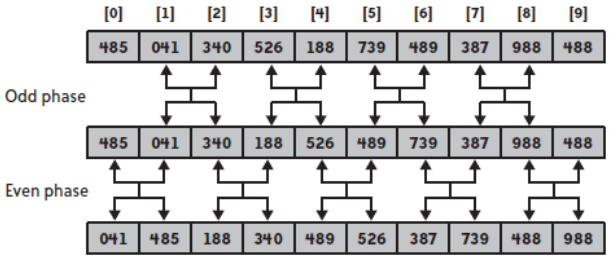

Fig. 1 Using odd-even transposition method

By odd-even transposition method we can sort *n* elements in *n* phases, each of which requires *n/2* compare-exchange operations. Although the complexity of this algorithm is still $O(n^2)$, but there is already increase of performance. In future chapters we will use this method and modify it again by combining with multithreading.

Now when we have the first parallel version of bubble sort we can experimentally measure the execution time of sequential and parallel algorithms, compute speedup. For getting more accurate results we will repeat the above operations for many times with different input lists that will be generated randomly. Speedup is approximately 2 (S = 1.8 – 2.3) independently from input list size.

First parallel version of bubble sort with using odd-even transposition method does not require any additional memory.

3.2. Parallelization of merge sort

Merge sort can be well parallelized due to use of divide and conquer method. Merge sort's original algorithm is demonstrated in Fig. 2

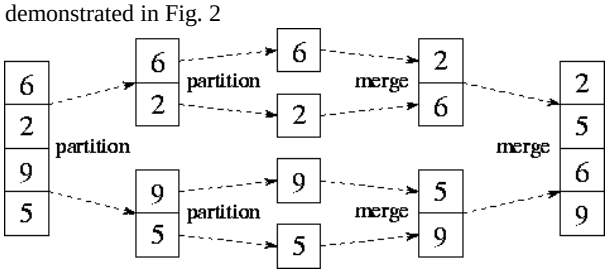

Fig. 2 Merge sort algorithm demonstration

As the algorithm is implemented through recursion, we can use a new thread for each new recursive call. That is all dividing operations will be executed in parallel as it is showed in [1]. After synchronizing division threads a merge operation is performed sequentially. But this method does not give impressive speedup. If sequential merge sort required *log(n)* times *n* operations, here that operations are performed in parallel and we have complexity of *n*. So speedup here is only *log(n)*. We can conclude that the bottleneck here is the merge part which is performed sequentially. So we need to parallelize merging. Merging can be parallelized by using nested parallelism as it is showed in [1]. Schematically parallel merging is represented in Fig. 3.

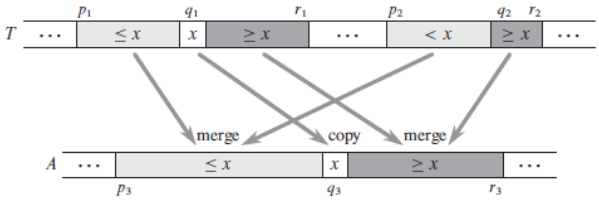

Fig. 3 Parallel merge

Assume that we need to merge subarrays $T_1[p_1 \ldots r_1]$ and $T_2[p_2 \ldots r_2]$ into the subarray $A[p_3 \ldots r_3]$. Letting $x = T[q_1]$ be the median of $T_1[p_1 \ldots r_1]$ and q_2 be the position in $T_2[p_2 \ldots r_2]$ such that x would fall between $T_2[q_2 - 1]$ and $T_2[q_2]$, every element in subarrays $T_1[p_1 \ldots q_1 - 1]$ and $T_2[p_2 \ldots q_2 - 1]$ is less than or equal to x, and every element in the subarrays $T_1[q_1 + 1 \ldots r_1]$ and $T_2[q_2 + 1 \ldots r_2]$ is at least x. To merge, we compute the index q_3 where x belongs in $A[p_3 \ldots r_3]$, copy x into $A[q_3]$, and then recursively merge $T_1[p_1 \ldots q_1 - 1]$ with $T_2[p_2 \ldots q_2 - 1]$ into $A[p_3 \ldots q_3 - 1]$ and $T_1[q_1 + 1 \ldots r_1]$ with $T_2[q_2 \ldots r_2]$ into $A[q_3 + 1 \ldots r_3]$. And as it is shown in [1] theoretically parallel merge sort is $O(n/\log(n^2))$ faster than sequential merge sort.

Although we reached $O(n/\log(n^2))$ theoretical parallelism, it is not working well in practice. The reason of that is in thread count. It is apparent that in parallel version of merge sort huge amount of threads are created which leads to CPU and memory resource consumption and reduction of effectiveness. So it would be nice to limit depth of recursion, which will limit the number of creating threads. That can be reached in two ways. The first is to define a threshold size of a list and if in some level of recursion the size of list is less than the defined size run sequential algorithm instead of parallel. But this is not a complete restriction, because if the input list is quite big than our defined threshold size then we will get a lot of threads. To prevent this behavior we can supply the second restriction – that is restriction through depth. We can pass depth as a parameter to merge sort

function. On each recursion step it will be incremented and when it becomes equal to zero we will finish recursion. Depth should be chosen based on physical thread count of a computer. Practical graphs of speedup as a function of input list is shown in Fig. 4. Here threshold is chosen 1024 and computer has 2 cores with hyper threading.

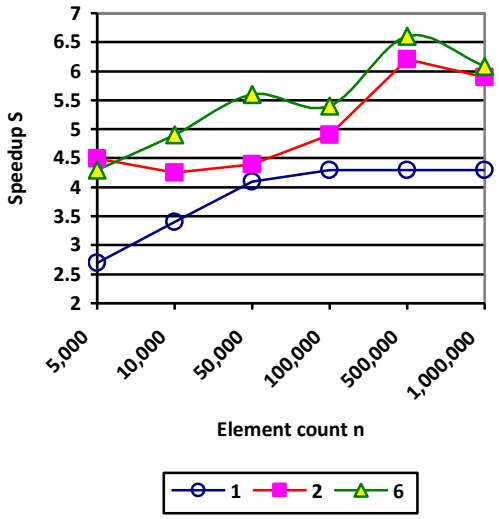

Fig. 4 Parallel merge sort performance. Speedup as a function of element count

In Fig. 4 there are three graphs and each of them shows speedup for depth 1, 2, 3. The graph in case of depth 4 is quite like the graph with depth of 3. Beginning from depth 5 speedup starts to decrease. So for this computer optimal depth would be 3 or 4. Fig. 4 shows that the algorithm is more effective for larger lists.

This parallel version of merge sort requires additional memory as much as input list. It uses also number of threads but it is worth for time critical tasks.

4. PARALLELIZATION THROUGH MULTITHREADING

Threading provides a mechanism for programmers to divide their programs into more or less independent tasks with the property that when one thread is blocked another thread can be run [2]. There can be several approaches by using multithreading parallelization. Approaches can be specific to a sorting algorithm or generic for all sorting algorithms. For specific approaches complexity of the new parallel algorithm is also very specific.

As an example of general approach can be - dividing input list into several smaller lists, sort each new list in a new thread and merge them into the final output list. In this case complexity of a sorting algorithm is predictable. It can be represented as sum of copying, sorting, and merging complexities.

At the beginning there is a need to copy from input list into t smaller lists, where t is the number of threads. Each smaller list has size of n/t, where n is the size of our input list. Copying requires n iterations and has linear complexity $O(n)$.

Complexity of sorting each small list can be computed by replacing n with n/t in algorithm's complexity's formula. For instance as bubble sort's complexity is $O(n^2)$, complexity of sorting each small list would be $O(n^2/t^2)$.

Merging two lists into the third one requires n_1+n_2 steps, where n_1 and n_2 are the sizes of lists. In our case for each merge iteration we merge the current list with the size of n/t and output list the size of which is different for each iteration. On the first iteration we need to merge lists with sizes of n/t and 0, on the second iteration – lists with sizes of n/t and n/t, on the third iteration – lists with sizes of n/t and $2n/t$. On the last iteration we need to merge lists with sizes n/t and $(t-1)n/t$. Now we need to calculate the sum of all merges' steps. The size of output list on each iteration increases by n/t and can be represented as arithmetic progression with t terms $(0, n/t, 2\cdot n/t,\ldots, (t - \cdot)\cdot n/t)$. Total steps for all merges can be calculated as follow:

$$t\cdot\frac{n}{t}+\frac{1}{2}\left(0+\frac{(t-1)\cdot n}{t}\right)\cdot t = n+\frac{n}{2}(t-1)=\frac{n}{2}(t+1)$$

Finally we can say that the multithreaded parallel sorting algorithm's complexity can be represented as

$$O\left(n+O_{algorithm}+\frac{n}{2}(t+1)\right), \quad (5)$$

where $O_{algorithm}$ is algorithm specific complexity.

Multithreaded parallel bubble sort's complexity would be

$$O\left(n+\frac{n^2}{t^2}+\frac{n}{2}(t+1)\right). \quad (6)$$

It is worth mentioning that count of threads that can be executed simultaneously depends on the environment i.e. number of processors, hyper threading. It is clear from (6) that for large values of t (approximately to n) this multithreaded parallel version is inefficient, because complexity of merging becomes quadratic. Although the algorithm's complexity remains quadratic, but for optimal values of t and n it has high speedup. Graph of speedup as a function of thread count based on experiments is represented in Fig. 5.

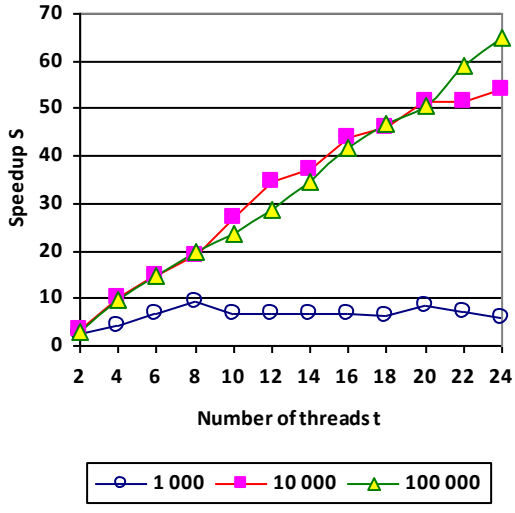

Fig. 5 Parallelization through multithreading. Speedup as a function of count of threads

Fig. 5 shows speedup as a function of count of threads for lists of sizes 1 000, 10 000, 100 000. It is clear from Fig. 5 that multithreaded parallel version of bubble sort works better for big lists. As a disadvantage of this version can be mentioned that it requires additional memory twice larger than input list.

This version also can be used mixed with bubble sort's

odd-even transposition method, which provides better results. Experimental graph of speedup as a function of count of threads in case of multithreaded bubble sort with odd-even transposition method is represented in Fig. 6.

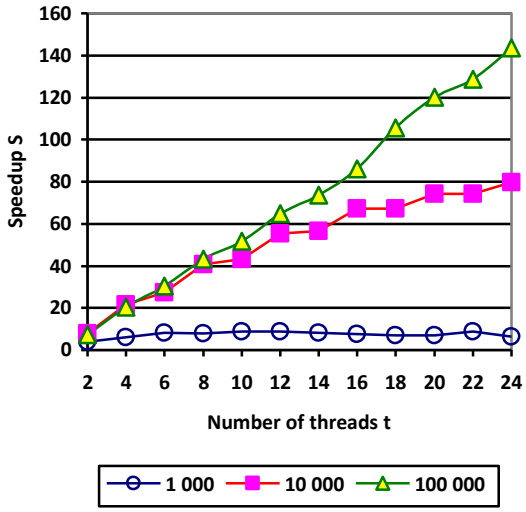

Fig. 6 Parallelization through multithreading using odd-even transposition method. Speedup as a function of count of threads

Comparing the results of Fig. 5 and Fig. 6 it is clear that the multithreaded bubble sort with odd-even transposition method is up to 2 times effective than just multithreaded bubble sort.

Also note that this approach is not effective for merge sort as main time consuming time for merge sort is final sequential merges.

5. PARALLELIZATION THROUGH OPENMP

OpenMP is a set of compiler directives, library routines, and environment variables that specify shared-memory concurrency in FORTRAN, C, and C++ programs [4]. OpenMP was first introduced in 1997. Basically all C++ compilers support the OpenMP language. OpenMP directives demarcate code that can be executed in parallel (called parallel regions) and control how the code is assigned to threads [4]. In many applications, a large number of independent operations are found in loops. Using the loop work-sharing construct in OpenMP, we can split up these loop iterations and assign them to threads for concurrent execution. The parallel for construct will initiate a new parallel region around the single for loop following the pragma and divide the loop iterations among the threads of the team. Upon completion of the assigned iterations, threads sit at the implicit barrier at the end of the parallel region waiting to join with the other threads [4].

As in multithreaded parallelization in parallelization through OpenMP also can be specific and general approaches. For example if there are independent operations in loops of a sorting algorithm, it can be parallelized by using loop work-sharing construct of OpenMP. That would be specific to a sorting algorithm. As an example of general approach can be - dividing input list into several smaller lists, sort each new list in independently by using loop work-sharing construct and merge them into the final output list. This method is like the previously presented method with multithreading.

In case of bubble sort simple loop work-sharing construct will not work, because of loop operations dependencies. So there can be two approaches here – either use the second general method described above or use the loop work-sharing construct with bubble sort's odd-even transposition modification.

The first general method is much like the bubble sort's multithreaded version and we will not go deep into it. The only difference is that here each small list is sorted as a separate loop operation. Experimental graph of speedup as a function of thread count is represented in Fig. 7.

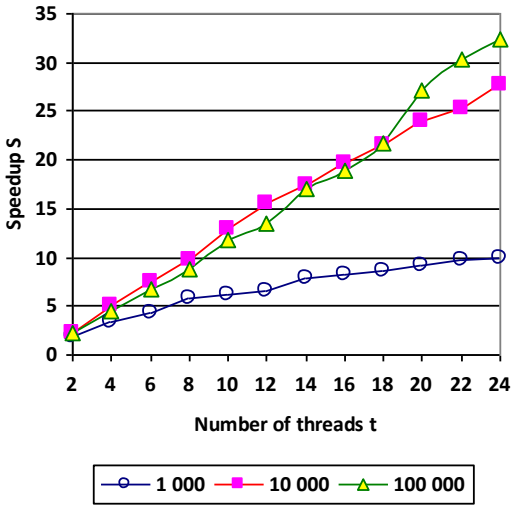

Fig. 7 Parallelization through Open MP. Speedup as a function of count of threads

As in Fig. 5, Fig. 6 as well as in Fig. 7 parallel algorithm is more effective for large lists. For the second method we will use bubble sort algorithm's modification represented in paragraph III. As it is shown in [4] for a common list, the parallel version of bubble sort will perform n iterations of the main loop and for each iteration a number of $n-1$ comparisons and $(n-1)/2$ exchanges will be performed in parallel. For this version we let OpenMP choose the number of threads. As a rule in that case the thread number equals to physical thread count of processors. If the number of threads is lower than $n/2$, every processor will execute $(n/2)/t$ comparisons. In this case the complexity of the algorithm will be $O(n^2/2t)$. If the number of physical threads is higher than $n/2$, the complexity level of the inner loops is $O(1)$ because all the iterations are performed in parallel. The main loop will be executed for maximum n times. So we can conclude that the complexity level of the algorithm is linear - $O(n)$, which is better than conclude that the complexity level of the algorithm is linear - $O(n)$, which is better than $O(n \log n)$, the complexity of the fastest known sequential sorting algorithm. As a result – theoretically bubble sort can have complexity of $O(n)$ if the number of physical threads equals the size of input list. But this is not much practical, because in practice the lists that need to be sorted are many times larger than the count of physical threads of a computer.

Note that this approach is also not effective for merge sort as main time consuming time for merge sort is final sequential merges.

6. CONCLUSION

In this work we represented several methods of parallelization of sorting algorithms and implemented some

of them on the example of bubble sort and merge sort. By using odd-even transposition method we acquired speedup approximately 2 for lists of all sizes. Multithreaded version of bubble sort for large lists provides speedup up to 70. Multithreaded version of bubble sort with odd-even transposition method is more effective and provides speedup up to 140. Parallelization through OpenMP is also more effective with large lists, it provides speedup up to 35. But this versions require two times bigger additional memory than the input list's memory. If the count of physical threads equal the input list's size linear complexity can be reached by using odd-even transposition method with OpenMP.
Parallelization of merge sort provided speedup up to 6.5 times in case of depth 3 or 4. For merge sort pattern is basically the same – it is more effective for larger lists.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, "Introduction to algorithms," 3rd ed., Cambridge, Massachusetts, London, England, The MIT Press, 2009

[2] P. S. Pacheco, "An Introduction to parallel programming", University of San Francisco, 2011, pp. 1-3, 61-62

[3] A. J. Bernstein, "Analysis of programs for parallel processing", *IEEE Trans. Electronic Computers*, vol. 15, pp. 757-763, Oct. 1966

[4] C. Breshears, "The art of concurrency", O'Reilly, 2009

[5] B. Chapman, G. Jost, R. van der Pas, "Using OpenMP: Portable shared memory parallel programming", Cambridge, Massachusetts, London, England, The MIT Press, 2008 year.