Global Journal of Computer Science and Technology
Hardware & Computation
Volume 12 Issue 10 Version 1.0 Year 2012
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Improving of Quicksort Algorithm Performance by Sequential Thread or Parallel Algorithms

By Abdulrahman Hamed Almutairi & Abdulrahman Helal Alruwaili
King Saud University

Abstract - Quicksort is well-know algorithm used for sorting, making O(n log n) comparisons to sort a dataset of n items. Being a divide-and-conquer algorithm, it is easily modified to use parallel computing. The aim of this paper is to evaluate the performance of parallel quicksort algorithm and compare it with theoretical performance analysis. To achieve this we implement a tool to do both sequential and parallel quicksort on randomly generated arrays of different size in several runs to provide us with enough data to draw conclusions about the efficiency of using the capability of modern multicore processors together with algorithms designed to increase the speed of sorting large arrays.

Keywords : *Parallel computing, Parallel algorithms, Parallel Computing, Quicksort.*

GJCST-A Classification: *F.2.2*

Strictly as per the compliance and regulations of:

© 2012. Abdulrahman Hamed Almutairi & Abdulrahman Helal Alruwaili. This is a research/review paper, distributed under the terms of the Creative Commons Attribution-Noncommercial 3.0 Unported License http://creativecommons.org/licenses/by-nc/3.0/), permitting all non-commercial use, distribution, and reproduction inany medium, provided the original work is properly cited.

Improving of Quicksort Algorithm Performance by Sequential Thread or Parallel Algorithms

Abdulrahman Hamed Almutairi [α] & Abdulrahman Helal Alruwaili [α]

Abstract - Quicksort is well-know algorithm used for sorting, making O(n log n) comparisons to sort a dataset of n items. Being a divide-and-conquer algorithm, it is easily modified to use parallel computing. The aim of this paper is to evaluate the performance of parallel quicksort algorithm and compare it with theoretical performance analysis. To achieve this we implement a tool to do both sequential and parallel quicksort on randomly generated arrays of different size in several runs to provide us with enough data to draw conclusions about the efficiency of using the capability of modern multicore processors together with algorithms designed to increase the speed of sorting large arrays.

Keywords : Parallel computing, Parallel algorithms, Parallel Computing, Quicksort.

I. Introduction and Motivation

Sorting is among the fundamental problems of computer science. Sorting of different datasets is present in most applications, ranging from simple user applications to complex software. Today, in this modern age, the amount of data to be sorted is often so big, that even the most efficient sequential sorting algorithms become the bottleneck of the application. It may be a database or scientific data.

Today, it is said that the problem is not the lack of data but the need to sort and search in the huge amount of data available to us. To be able to do those tasks efficiently with the data available, the speed of sorting becomes critical. A lot of work was done to improve the speed of traditional sequential sorting algorithms, be it optimizing the pivot selection for quicksort or trying to come up with new, adapting sorting algorithms.

With the appearance of parallel computing, new possibilities have appeared to remove this bottleneck and improve the performance of known sorting algorithms by modifying them for parallel execution. At first, this was achieved using distributed computing, however with the hardware available today, it is possible to do this even on home computers thanks to their multicore processors.

In this paper, we present a parallel version of the well know quicksort algorithm and compare its performance to the performance of its simpler, sequential quicksort algorithm. Comparing their performance, we look for the threshold T, the size of the array, at which the parallel algorithm becomes actually slower than the sequential algorithm. By choosing a value that promises the best performance, we then test and compare the parallel and sequential versions of the quicksort algorithm, providing us with enough data to draw a conclusion about the increase in performance when using the parallel quicksort algorithm.

Author α : King Saud University, Master of Computer Science Program-Parallel Processing Course , Dr.Abdullatif Alabdullatif.

II. Quicksort

Quicksort is a sorting algorithm developed by Tony Hoare that requires, on average, O(nlogn) comparisons to sort n items. In the worst case scenario, it makes O(n2) comparisons, even though this is a rare occurrence. In reality it is mostly faster than other O(nlogn) algorithms [1]. The implementation of a simple sequential Quicksort algorithm follows that we choose for our needs is:

- Choose a pivot element. We use the last element out of the sorting area
- Iterate through the sorting area, placing all numbers smaller then the pivot to a position on its left, while placing all other numbers to a position on its right. This is done by swapping elements.
- The pivot is now considered to be in its sorted position and we continue with the divide-and-conquer strategy, applying the same algorithm on the part to the left and the part to the right of the pivot.

This way, the whole, original dataset is sorted by recursively using the same algorithm on smaller and smaller parts. This is done sequentially. However, once the partitioning is done, the sorting of the new sorting areas can be performed in parallel as there is no collision.

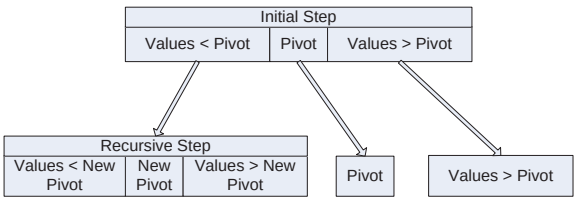

Figure 1: Simple graphical representation of the Quicksort algorithm

III. Related Work

Several works were done into parallel sorting algorithms. The first one, being restricted by hardware not providing multicore processors were using private

© 2012 Global Journals Inc. (US)

virtual [3]. This required the solutions to communicate between them using messaging witch not only complicated the process of implementation, but also increases the overhead of parallelism. The work presented a fine-tuned parallel quick sort. The algorithm used asynchronous multiprocessors with cache-coherent shared memory. They pick the pivot using the processor with the lowest ID. Afterwards, each processor picks a blok from the left and one block from the right side of the pivot, the size of the block chosen so it can fit into the L1 cache. This two block processed in a way so that one block contains values smaller then the pivot, and the other block contains values larger or equal, with again the CPU with lowest ID performing some after-cleanup operations. Then the input array is distributed between the processors and the recursion sorting of quicksort begins until each group of processors contains only one CPU or until the size of arrays are below a certain limit and insertion sort is performed. Here, we found especially the idea of a threshold to use a simpler, sequential sorting algorithm to save the computational cost when sorting small arrays, especially interesting.

However, in last few years, improvements in sorting have been made thanks to the works incorporating multi core and multiprocessor computer architecture [2].

IV. Theoretical Analysis

With the parallel algorithm, we have to remember that the cost of creating, monitoring and managing of the parallel tasks is added to the total computational cost. Let's assume the average case of quicksort with computational time O(nlogn).

When using parallel computing, the computational cost consists of these values:

- picking the pivot – O(1)
- moving the elements to the left and right side of pivot - O(n)
- creating new Tasks to sort the left and right part - O(1)

Based on Figure 1 it's easy to see, that the fully developed parallel quicksort algorithm will have the shape of a binary tree.

For each leaf node of this tree, we will be required to perform a sequential quicksort algorithm, the size of the leaf node depending on the threshold T we choose.

For each node, the creation of new Tasks for child nodes will add to the total computational cost.

The extreme condition would be where the last leaf node would be smaller then T in case N is not divisible by T. However, this has an minimal impact on the overall performance and therefore we decided to assume, which allows us to make our theoretical analysis using a complete binary tree.

For a dataset of N elements, the binary tree will have N/T leaf nodes. Therefore it can be easily seen, that the tree will have nodes.

The number of the leaf nodes will be N/T, each of it with the size of T. This means, the computational cost to sort the leaf nodes using sequential quicksort will be With this in mind, if we would ignore any overhead, parallel quicksort would be able to provide us with this increase in performance, as shown in Figure 2. In theory, lower threshold values would provide us with even better performance.

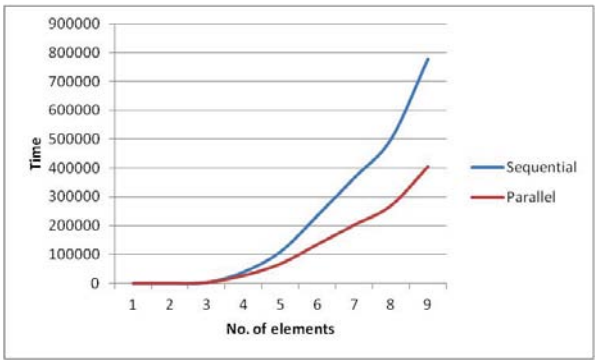

Figure 2 : Theoretical comparison between Sequential and Parallel quicksort, using threshold value of 500

However, given the binary tree, in each node we have to select a pivot, move the elements to the left and right side of the pivot and create the Tasks to do the parallel sorting.

The limitation to speed increase of a parallel algorithm as compared to a sequential algorithm are the overhead caused by the need to create new, parallel processes and their management.

V. Practical Analysis

For our practical analysis, we wrote a simple program to test and compare a sequential and parallel quicksort. This program first sorts a field of integers using sequential quicksort and then sorts the same field using parallel quicksort. To evaluate the overhead needed to create new tasks, we implemented a version of the quicksort algorithm, where for each recursive call, a new task is created, they are however executed not parallel but sequential.

Our solution was implemented in C# as a simple form application. The window can be seen in Figure 3.

© 2012 Global Journals Inc. (US)

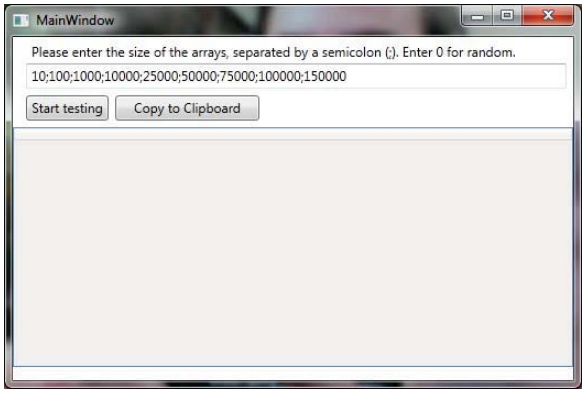

Figure 3 : The application used to compare the performance sequential and parallel quicksort

From the point of user interface, the application is very simple, providing 4 items for user interaction. The first one is a textfield where wecan define the size of arrays to be used for the testing. Next is the 'Start testing' button which begins the sorting process. The last button is the Copy to clipboard button which takes the aggregated results of the tests run so far and copies them into a table structure in the clipboard, which can be easily pasted into Excel for further processing.

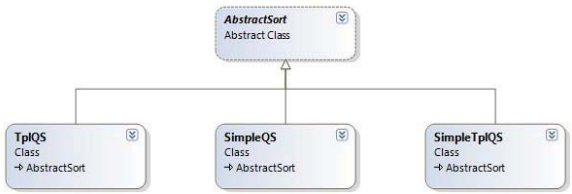

Figure 4 : The core class design of the benchmarking program

The base class is the AbstractSort, which contains the code for benchmarking the sorting. The derived classes override an abstract method called InnerSort and implement either the sequential or parallel version of quicksort.

This is the code for the public method Sort of the AbstractSort called.

```
int[] sorting=new int[array.Length];
Array.Copy(array, sorting, array.Length);
QSResult result=new QSResult();
DateTime start= System.DateTime.Now;
result.sorted = InnerSort(sorting);
result.time =
System.DateTime.Now.Subtract(start).Total
Milliseconds;
return result;
```

As can be seen, a copy of the array is created so that when the other sorting algorithm is to be benchmarked, it will be benchmarked on the same, unsorted array. Afterward the current time is stored and the sorting commences. When the sorting is finished, the result time is computed as current time minus the time stored before the sorting commenced.

The sequential quicksort is implemented in a simple manner as the following pseudo code shows:
qsort(array,left,right)

```
qsort(array,left,right)
{
  cur, last;
  if (left >= right) return;
swap(array,left,(left+right)/2);
last = left;
for(cur=left+1;cur<=right;++cur)
  {
    if(array[cur]<array[left])
    {
      ++last;
     swap(array,last,cur);
    }
  }
  swap(array,left,last);
qsort(array,left,last-1);
qsort(array,last+1,right);
}
```

Our parallel implementation is only slightly different. First, the size of the array to be sorted is checked against a threshold. If the size is smaller than the threshold, sequential quicksort is used as the overhead of creating new tasks would slow the sorting process too much. If the size of the array is bigger than the threshold, instead of calling the quicksort for each part directly, new Task is created for each of the call and let's them handle the sorting of each part of the array. This is show in the next code snippet:

```
if ((last - left) < SEQ_THRESHOLD)
{
       qsort(array, left, last - 1);
       qsort(array, last + 1, right);
}
else
{
       Task.WaitAll(
       Task.Factory.StartNew(()=>
       qsort(array,left,last-1)),
```


© 2012 Global Journals Inc. (US)

```
            Task.Factory.StartNew(()=>
            qsort(array, last + 1, right)));
        }
```

Last, to test the overhead caused by creating a new Task, we created a quicksort algorithm where new Task is created for the initial sorting. This is done by the following code when initializing he quicksort algorithm:

```
Thread thread=new Thread(delegate()
{
        qsort(array, 0, array.Length - 1);
});
thread.Start();
while (thread.IsAlive)
    Thread.Sleep(1);
```

These algorithms were tested on the same hardware, using a quad core processor.

To test we used 9 randomly generated arrays of following sizes: 10; 100; 1 000; 10 000; 25 000; 50 000; 75 000; 100 000; 150 000. We did 100 separated runs, algorithms in each run using the same data, but the data being randomly generated between runs to provide variability.

The hardware we run our test on was a Intel Core i5 processor running at 2.53GHz. This is a dual core processor capable of running four threads in parallel. The computer had 3.8GB of memory. During the test, no other program was running to provide a interference-free environment.

First, by using a trial and error approach, we established a suitable value for the SEQ_THRESHOLD value to be 1000. We ran a full scale test on arrays of 9 different sizes with three different SEQ_THRESHOLD values, 1000,5000 and 50 000. The resulting times can be seen in Table 1 and in Figure 5.

	T=1000	T=5000	T=50000
10	0.01	0	0.010001
100	0.01	0.020001	0.050004
1000	0.250016	0.270011	0.260018
10000	2.010118	2.880166	3.060169
25000	5.380318	6.120344	9.15052
50000	11.36065	11.320644	19.61112
75000	18.14103	18.251045	28.60164
100000	24.91142	22.591294	34.19196
150000	36.41208	34.551976	46.99269

Table 1: Average run times for different threshold and number of elements

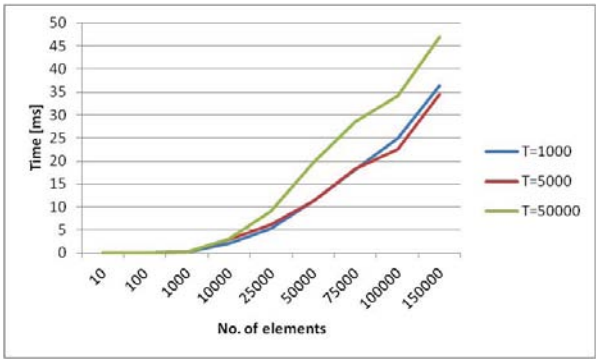

Figure 5 : Comparison of parallel quicksort run time with different thresholds

With the threshold set at 50000, the parallel algorithm is actually slower, as the computational cost of creating new tasks increases the total run time, but the parallelism is not utilized enough to offset this. In Figure 6 it can be clearly seen, that with the higher threshold of 50000, the additional computation cost of creating new threads cannot be compensated for by doing parallel computation as the algorithm returns to sequential quicksorting too soon.

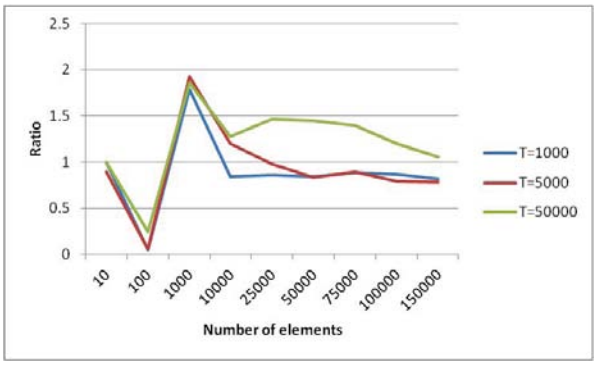

Figure 6 : Ratio of run time, Parallel to Sequential

We can see the comparison of a sequential and parallel quicksort in Figure 7.

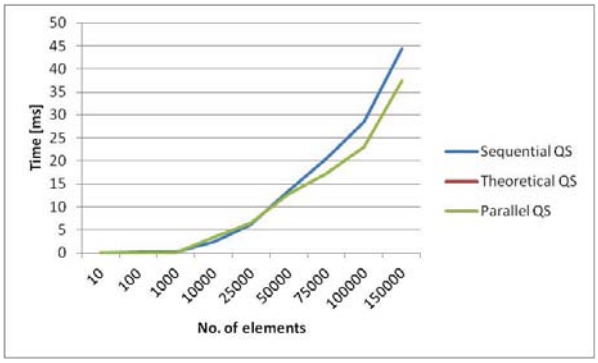

Figure 7 : Comparision of sequential and parallel quicksort, T=1000

To compare the speed gained by using parallel computing, we created a graph showing the speed up ratio for different data size as shown in Figure 8.

© 2012 Global Journals Inc. (US)

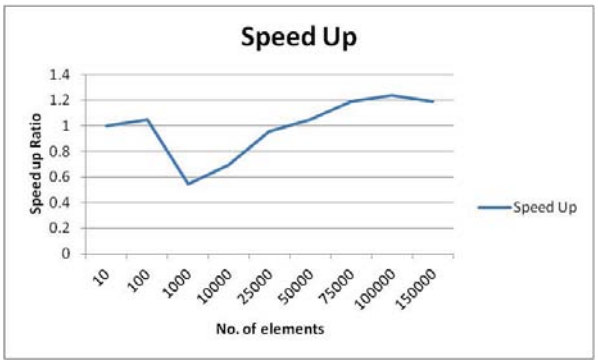

Figure 8 : Graph showing speed up ratio by using parallel quicksort

This graph was created by using the formula

$$Speedup = \frac{execution\ time\ of\ sequential\ algorithm}{execution\ time\ of\ parallel\ algorithm}$$

As can be seen here, at low number of elements no speed up is achieved. As the number of elements increases, the speed of sorting actually decreases. This is caused as stated before by the overhead needed for creating the parallel tasks and as there is not enough elements for the parallelism being able to compensate for this. After the number of elements increases enough, the overall speed and speed gain increases as well by about 20%.

VI. Conclusion

We successfully implemented a parallel version of quicksort algorithm. After choosing a appropriate threshold value to switch from parallel to sequential sorting, we observed the performance of the algorithm. The results are obviously in favor of the parallel quicksort algorithm. Using a reasonable threshold to return to sequential quicksort, we are able to circumvent the increased computational cost of creating new tasks for small datasets, while with the bigger datasets we take advantage of the parallelism possible by today's hardware. And thanks to simplicity of the parallel implementation of quicksort algorithm it is easy to achieve major, 20% increase of performance when sorting a larger dataset.

References Références Referencias

1. S. S. Skiena, The Algorithm Design Manual, Second Edition, Springer, 2008, p. 129
2. P. Tsigas, Y. Zhang, A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000
3. Z. Chengyan, Parallel Quick sort algorithm with PVM optimization, 1996

© 2012 Global Journals Inc. (US)

This page is intentionally left blank

© 2012 Global Journals Inc. (US)