The Time Profit Obtained by Parallelization of Quicksort Algorithm Used for Numerical Sorting

Valma Prifti
PUT
Tirane, Albania
valmaprifti1@gmail.com

Redis Bala
PUT
Tirane, Albania
redis@mail.com

Igli Tafa
PUT
Tirane, Albania
itafaj@gmail.com

Denis Saatciu
UT
Tirane, Albania
denis.saatciu@gmail.com

Julian Fejzaj
UT
Tirane, Albania
jfejzaj@gmail.com

***Abstract*—This paper presents an experimental description of how to use OpenMP for achieving high performance from the quicksort algorithm through the parallelization of some key sections of its code. When this work was in process, I was unsure if the time profit I would achieve would be evident, but I think I exceed my expectations in this matter. The only problem I faced was the unpredictability of execution time in some cases for the parallel version. Anyway, even in this case the time profit in comparison with the sequential version was clear and evident.**

***Keywords*—OpenMP; Quicksort; parallelization; numerical sort**

I. Introduction

As known, today almost all commercial processors are multicore and OpenMP library provides a way of allowing programmers to make use of of the parallelism offered by these processors. When OpenMP was first presented in 1997 it was said that it will sharpen the trend in which scientists and engineers head toward high-end workstations instead of using supercomputers for complex applications.[10] But no one knew that 16 years later every ordinary computer user would have in their hands a multicore computer with the attributes of a small workstation.

Since then, OpenMP has been accepted and used by a great number of developers but it has competed with POSIX threads (Pthreads) at one hand and Message Passing Interface (MPI) at the other. These 2 other methods are lower-level models of parallel programming but they cost the developer a lot more initial trouble in contrast with OpenMP which offers a much more simple API for writing parallel programs[1]. But we must be aware that not every sequential program can be transformed into an equivalent parallel program. OpenMP in practice is an API, which is implemented through library subroutines and a set of compiler directives.

Each thread has its private memory and its stack. All threads have the right to access the shared memory which can be used as a bridge of communication between them. The work is distributed among threads only for the parallel sections of the code, using the fork-join model.

The program starts with only one thread called the master thread. Then this thread is forked in a number of threads specified by the user or by the code. Each thread does its private work, using its private memory with its own private variables. When the parallel section comes to an end, all the parallel threads are joined into a single thread, recreating the master thread.

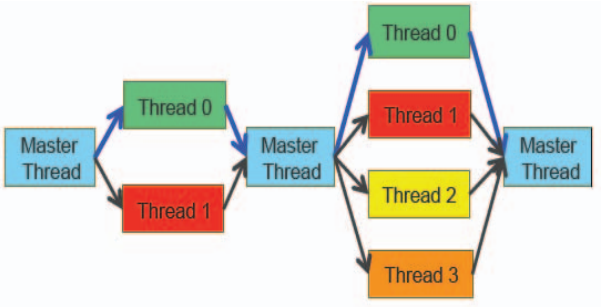

Fig. 1. Open MP flow of execution

The colors in fig.1 indicate that thread 0 of each parallel region is the direct descendant of the master thread.[9] The main problem that arises from this model of programming is the synchronization of the parallel threads. It would be a solved problem if these threads would finish their work simultaneously but this is not possible because each core of the machine has a scheduling list which must be followed strictly and the load may not be equal in each core. If one thread ends way earlier than another, it will have to wait causing the core to have idle cycles, therefore having a poor efficiency.[2] To overcome this problem OpenMP offers barriers and synchronization features.

II. Related Works

Various methods have been developed and proposed to parallelize sorting algorithms and in specialty the quicksort algorithm. Parallel sorting algorithm implementations have been addressed in various works, but all these parallel methods can be categorized in 3 major categories: OpenMP,

Pthreads (or Posix threads) and MPI (Message Passing Interface). One of the main works which also inspired me to this article is the paper [6]. In this paper there is a lot of detailed optimization of quicksort algorithm, even for optimizing the worst case scenario by choosing in a wise way the pivot element. His work is concentrated in Pthreads which is a much lower programming model than OpenMP and of course gives much control over resources and detailed timing behavior.[6] But this was not the intention of my work here. We were inspired by the simplicity that OpenMP offers for providing parallel execution and at the sime time helps efficient utilization of resources (cores in such case).

His work also gives a version of quicksort implemented in cluster systems which can be in different computers, even with different operating systems and they manage to communicate through message passing techniques offered by MPI. This gave him the advantage to test his implementation with a much bigger number of cores and with much bigger arrays to be sorted. We see that as the main reason why he choose a random number generator for initializing the array which is to be sorted.

Another work that we found interesting and would like to mention is the paper [11]. This work was concentrated on embedded environments and that's why limited resource usage was of a key importance in their paper. We were enchanted by the detailed schematics of timing behaviors in the execution of the program with different number of cores, and the performance diagrams presented in their work.

We strongly believe in the power of OpenMP, its simplicity and its rich library which grows every year offering more and more spaces to be used and new capacities. That is the main reason why I choose OpenMP to do this experiment.

III. Theory of Experiment

Regarding to the sorting algorithms, we guess it`s unnecessary to mention their importance in computers and how much effort programmers have put towards their evolution and optimization. Even for sorting our files in the file systems, special sorting algorithms are used depending from the OS. From all the known sorting algorithms, quicksort is the one that performs better in practice, despite other variants that give better results in papers but not in real life.

Quicksort belongs to the sorting algorithms based on divide-and-conquer logic, for sorting a random array of numbers. It has 3 phases in which each array A[l..r] passes before sorting:

- Divide – Array A[l..r] is divided in 2 subarrays (initially empty): A[l..m-1] and A[m+1..r], in such a way that every element of A[1..m-1] is smaller or equal to A[m], and every element of A[m+1..r] is bigger than A[m]. In this phase is chosen the index m, which is called the pivot element.
- Conquer – Sorts each subarray, A[l..m-1] and A[m+1..r] through recursive calls of quicksort.
- Combine – We now have 2 sorted subarrays which have to be combined together to form the sorted array A[l..r].[

The quicksort pseudo-code is as follows:

```
QUICKSORT(A, l, r)
1  if l<r
2     then m←PARTITION (A, l, r)
3        QUICKSORT (A, l, m-1)
4        QUICKSORT (A, m+1, r)
```

But the key part of this algorithm is the PARTITION procedure, which rearranges the array A[l..r]. The pseudo-code of PARTITION procedure is as follows:

```
PARTITION(A, l, r)
1  x←A[l]
2  i←p-1
3  for j←p to r-1
4     do if A[j]≤x
5        then i←i+1
6             exchange A[i] ⟷ A[j]
7  exchange A[i+1] ⟷ A[r]
8  return i+1
```

As we se, the PARTITION procedure selects a **pivot** element, which is used to partition the array A[l..r]. In our implementation, the choose pivot element will be always the first element of the array. The pivot element is the axis of the whole sorting algorithm.[3]

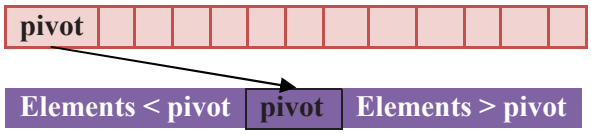

Fig. 2. Partitioning logic

After this procedure is done with the first array it continues recursively with the 2 subarrays created (the subarray with elements smaller than pivot and the subarray with elements bigger than the pivot). In every new subarray created, a new pivot element is selected. After the first step, the PARTITION procedure creates 4 regions in the array being sorted:

Fig. 3. Partitioning regions

When this procedure continues recursively until the subarrays created are with one element, all it is left to be done is combining these subarrays.

We should emphasize that the pivot elements selected in each subarray created are used only for dividing the elements

in the 2 subarrays, their position is not part of a comparison , but it is a result of the last move of this comparison.

This property of quicksort which divides each array into 2 subarrays is the reason why quicksort can be parallelized without affecting its logic but at the same time benefiting execution time.[5]

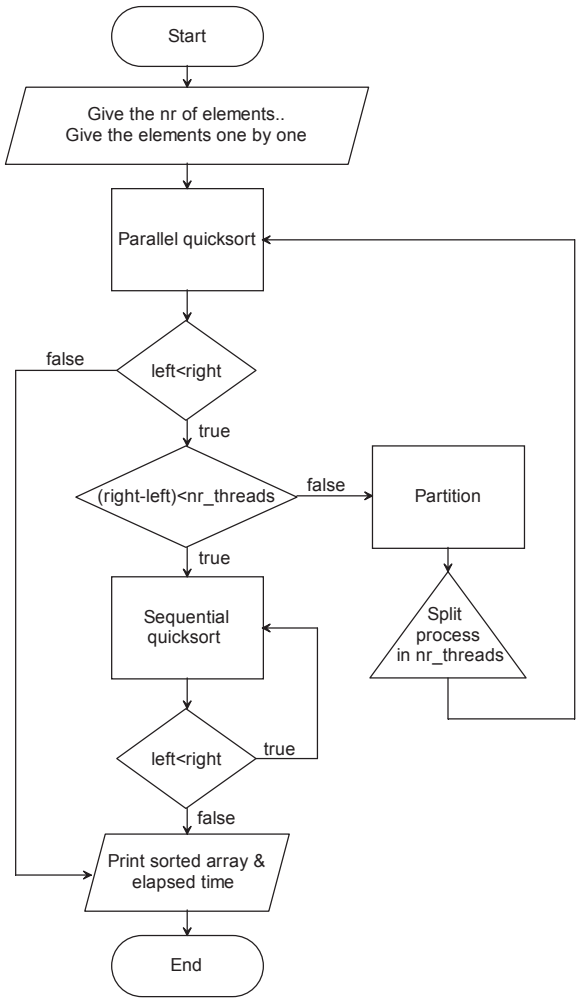

Fig. 4. Parallel quicksort flowchart diagram

For exploiting processor cores (or threads in case of Hyperthreading technology) in the most efficient way possible, we must be very careful in choosing tasks for each thread in such a way that we avoid or minimize idle cycles of cores. As mentioned before, if we have less threads than blocks of code to be executed, some of the threads or all of them will have to execute more than one blocks of code. If we have less blocks of code than threads the remaining threads will be idle so we will have poor efficiency.[7] Here is where OpenMP shows its power, letting the programmer to determine the distribution of load in independent units.

Using the directive **#pragma omp parallel sections** we will identify the section of the code which will be divided between the threads created.

Then the #pragma omp section specifies in a more explicit way the blocks of code which will be executed by each individual thread.[4]

The reason why we have specified the variables: A, left, middle and right as **firstprivate** is because these variables must be initialized with the value they had before the parallel section. After exceeding the parallel section, these variables will have the same value they had before entering in this section.[8] This is very important because me want to maintain the borders of the initial array, but at the same time we have to work with subarrays which are small pieces of the initial array.

IV. Experimental Phase

The operating systems which we choose as environment for executing and testing my experiment is Linux OpenSUSE 12.3. The reason why we made this decision is because we was very familiar with it and because OpenSUSE 12.3 comes with OpenMP library preinstalled for gcc and g++ which saved me a lot of trouble. The hardware used is an EliteBook workstation with dual-core CPU and 6 mega cache size.

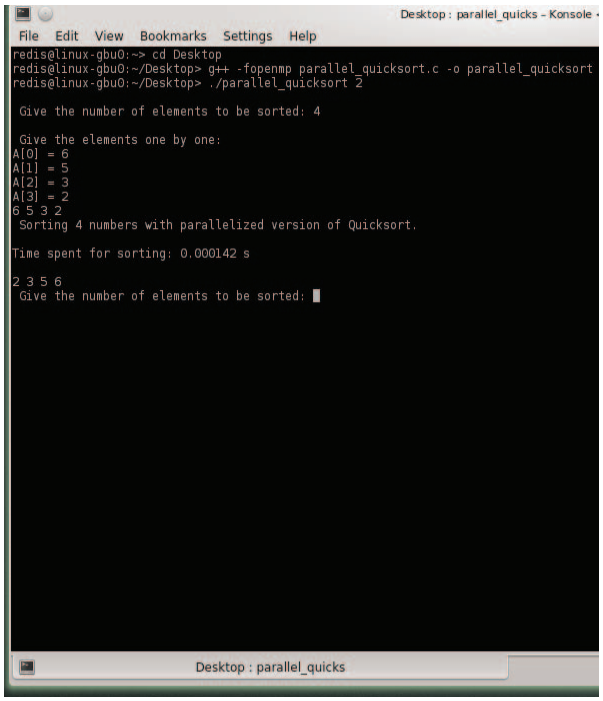

Fig. 5. Screenshot of my application

For compiling an OpenMP file we must open the terminal and write the following:

g++ -fopenmp quicksort.c –o quicksort

The instruction above produces the compiled version of our program, which is now ready for execution. For execution we must type the following in the terminal:

./quicksort 2

The number that follows the quicksort is the number of

threads we want to use for our program. This number is taken by the program as part of the command line arguments of the main function. If by mistake we do not specify the number of threads the execution will stop and the program will print the following:

Please specify the number of threads u want to use

When we have given the number of threads to be used by the program, we will be asked for the number of elements we want to sort. After that we have to input one-by-one each element of the array to be sorted. This can be a bit tiresome but I surely have an important reason for that. If we didn't insert each number one by one anyone would think that using a random number generator would be a smarter and less tiresome way. But this doesn't give us control over the numbers and their initial order. So it would be impossible for us to compare timing behaviors of sequential and parallel versions of the program knowing that the initial order of the element is crucial for the overall execution time of the program. For example, for our type of implementation which uses the first element as the pivot , when elements are given in ascending order, this takes longer time for sorting because is the worst case scenario.

TABLE I. SEQUENTIAL QUICKSORT TESTS

Test no.	Number of elements in the array	Execution time (in seconds)
1.	5	0.025
2.	10	0.034
3.	15	0.036
4.	20	0.049
5.	30	0.067
6.	40	0.071
7.	50	0.100
8.	60	0.103
9.	70	0.144
10.	80	0.246

TABLE II. PARALLEL QUICKSORT TESTS

Test no.	Number of elements in the array	Execution time (in seconds)
1.	5	0.000279
2.	10	0.000556
3.	15	0.000647
4.	20	0.000652
5.	30	0.000959
6.	40	0.001652
7.	50	0.000780
8.	60	0.002393
9.	70	0.002628
10.	80	0.003240

V. CONCLUSIONS

By interpreting the results of testing depicted in the tables above, which are derived from 3-5 iterations for each experiment we think is unquestionable the advantages that the parallel version has in comparison with the sequential version. Judging by the execution time, the parallel version has a slower precipitance in time extension during the same increase in the number of elements in the array. In other words, while the number of elements in the array grows, the sequential version will have a larger rate of time extension. Another important thing that is worth mentioning is that the execution time of the parallel version, in some cases showed some unpredictable results. This can be seen in the results of the test number 7 (marked with red). It would be logical that the execution time for this test should be somewhere between of the previous and the next testing results. But this didn't happen in practice. In fact we faced this problem many times while executing the parallel version. The main reason why this happened was because of the sequence in which we gave the numbers to be sorted. The worst case scenario would be if the numbers in the input array would be in the ascending order. This would give the longest sorting time. We tried to give the initial numbers in the worst case possible but sometimes we even randomized their sequence a bit, just to test the sorting accuracy. We noticed that even the load of the system in the moment of execution did affect the execution time. This was of course a consequence of the scheduling algorithm used by the OS.

We think that despite the unquestioning advantage in execution time of the parallel version, we must be very careful where we implement it. Because if the application we are about to build needs predictable sorting time, the parallel version may produce unwanted and odd results in execution time. But we think this problem is worrisome only for some special applications which I mentioned above.

VI. FUTURE WORKS

One of the main reasons why I started this work was because of my interest in parallel system. Their spreading in nowadays applications and their prespective of growth is very clear. But another model of parallelized systems are the computer clusters. In my future work I would like to try parallelizing this algorithm using the Message Passing Interface in a network of computers and the comparison of the time profit in comparison with this paper.

REFERENCES

[1] B. Chapman, G. Jost, R. Van Der Pas, Using OpenMP, Portable Shared Memory Parallel Programming, 2011

[2] N. Matloff, Programming on parallel Machines, University of California, 2009

[3] Introduction to Algorithms 2nd edition,

[4] Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein, OpenMP Application Program Interface Version 4.0, July 2013

[5] Taking Advantage of OpenMP 3.0 Tasking with Oracle Solaris Studio – An Oracle white paper, November 2010

[6] P. Kataria, Parallel quicksort implementation using MPI and Pthreads, 2012

[7] R. Chandra. L. Dagum, D. Kohr Dror Maydan, J. McDonald, R. Menon, Parallel Programming in OpenMP, 2010

[8] OpenMP 4.0 API C/C++ Syntax Quick Reference Card, OpenMP Architecture Review Board, 2013

[9] C. Evangelinos, Parallel Programming for Multicore Machines using OpenMP and MPI, MIT/EAPS, 2011

[10] M. J. Quinn, Parallel Programming in C with MPI and OpenMP, 20012

[11] K. J. Kim, S. J. Cho, J. W. Jeon, 11th International Conference on Control, Automation and Systems - Parallel Quick Sort Algorithms Analysis using OpenMP 3.0 in Embedded System, 2011.

APPENDIX

Below is the source code of the parallelized version of quicksort:

```
#include <stdlib.h>
#include <iostream>
#include <math.h>
#include <stdio.h>
#include <sys/time.h>
#include <omp.h>


void display_array (int* A, int n)
 { for (int i=0;i<n;i++)
     printf ("%d ",A[i]);
 }


int partition (int* A, int left, int right)
 {
    int p=A[left];
    int middle=left;
    int temp;

    for (int j=left+1; j<=right; j++)
        {
          if (A[j]<p)
             {
               middle++;
               temp=A[middle];
               A[middle] = A[j];
               A[j] = temp;
             }
        }
    temp = A[left];
    A[left] = A[middle];
    A[middle] = temp;

    return middle;
 }


void sequential_quicksort (int* A, int left, int right)
 {   int middle;
     if (left < right)
     { middle = partition (A, left, right);
       sequential_quicksort (A, left, right - 1);
       sequential_quicksort (A, middle + 1, right);
     }
 }
```

```
void parallel_quicksort (int* A, int left,int right, int nr_threads)
 { if (left < right)
   {
    if ((right - left) < nr_threads)
     { sequential_quicksort (A, left, right);
     }
    else
     { int middle = partition (A, left, right);

       #pragma omp parallel sections firstprivate(A, left, middle, right)
       {
         #pragma omp section
         parallel_quicksort (A,left, middle - 1, nr_threads);
         #pragma omp section
         parallel_quicksort (A, middle + 1, right, nr_threads);
       }
     }
   }
 }


int main (int argc, char *argv[])
 { int n, i, x, nr_threads;
   int choice;  int pos;
   double start, finish;
   if (argc != 2)
    { printf (" Please specify the number of threads u want to use\n");
      return 1;
    }
   nr_threads = atoi(argv[1]);
   do
   { printf ("\n Give the number of elements to be sorted: ");
     scanf ("%d",&n);
     int A[n];
     printf ("\n Give the elements one by one: \n");
     for (i=0; i<n; i++)
       { printf ("A[%d] = ",i);
         scanf ("%d", &A[i]);
       }

     display_array(A,n);

     printf ("\n Sorting %d numbers with parallelized version of Quicksort.\n\n", n);
     start = omp_get_wtime();
     parallel_quicksort (A, 0, n - 1, nr_threads);
     finish = omp_get_wtime();
     printf ("Time spent for sorting: %f s\n", (finish - start));
     printf ("\n");

     display_array(A,n);

   }while (true);
   return 0;
 }
```