CSCI 563 A - Parallel Computing
Final Project: Quick Sort Parallelization
Final Report

Vinh Le
Colorado School of Mines 2018
Computer Science

CONTENTS

I. INTRODUCTION

In the CSCI 563 - Parallel Computing course we learned about many different methods to take a serial program and get speed up by utilizing parallel and multi-threading techniques. For the final project I decided to chose Quick Sort for the algorithm that I will parallelize and CilkPlus as the parallelization technique. The code is written in C++ and it is ran on the eecs-hpc-1 server. The compile command and the run commands are:

```
g++ project_Vinh_Le.cpp -o project -fcilkplus -std=c++11
```

```
./project n
```

Where n is the number of elements that you want to test.

II. ALGORITHM

The algorithm that I have selected to speed up is quick sort. Quick sort is a systematic method to sort elements in an array. When utilized properly it can be faster than its alternatives, heap sort and merge sort. Quick sort uses a comparison sorting method. A pivot point is selected and compared to each element if the element is less then the pivot then it is added at the beginning of the array. The larger values will remain in place and then the pivot will replace the value at the next incremented point. In the next iteration the "front end" and the "back end" are recursively called and a new pivot is selected. There are three normal ways to select the pivot point. The pivot can be the first element, the last element, or can be randomly selected. On average the time complexity of quick sort is O(nlogn), the worst case is $O(n^2)$. The worst case will only happen if the selected pivot is the largest element or the smallest element in each iteration.

Figure 1 shows one implementations of the quick sort algorithm. In this case the last element is selected as the pivot point. Then each element is compared to it in order and if the element is, in this case, larger move the element to the end. After iterating through all the elements, where the check index is equal to the pivot index return the pivot and sort the two halves.

III. IMPLEMENTATION

For this project I created an array that started at zero and then incremented to the number of elements, n. After creating the incremental array, I used the random_shuffle function in the algorithm library to shuffle the array. The random_shuffle function utilizes the random seed, therefore I initialized the random seed using the srand function and feeding it time(NULL). After the array is shuffled the array is copied for all implementations.

A. Serial Implementation

For the serial implementation I created three functions. The first function is the quick sort function based off of the following pseudo code. The second function is the partition function that is included within the quick sort function. The third function a utility swap function that takes in 2 elements and changes the address of the value and is used in the partition code. Following is the pseudo code for the quick sort and partition.

B. Pseudo Code for Quick Sort

```
low = Starting index
high = Ending index
quicksort(array, low, high)
  if (low < high){

    //rearrange using partition function
```

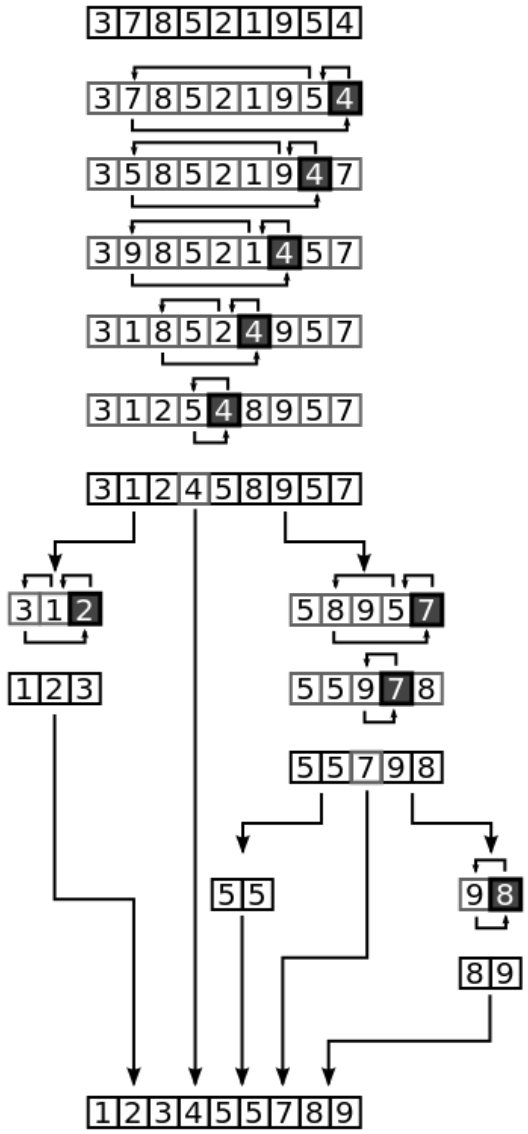

Fig. 1: Example of quick sort comparison and partition

```
    pi = partition(arr, low, high);

    //recursively call quicksort
    quicksort(arr, low, pi-1);
    quicksort(arr, pi + 1, high);
```

C. Pseudo Code for Partition

```
partition(array, low, high)
    //set high as pivot
    int pivot = arr[high];
    int i = (low - 1); //Set i index

    //iterate through array within index
    for (int j = low; j <= high- 1; j++)
        //if low switch with lowest index
        if (arr[j] <= pivot)
            i++; //increment i
            swap(&arr[i], &arr[j]);

    //swap pivot with next lowest
    swap(&arr[i + 1], &arr[high]);
```

D. Parallel Implementation

For the parallel implementation, I needed to first include the cilk library using the cilk/cilk.h include command and then use the keywords cilk_for, cilk_spawn, and cilk_sync. I used cilk_for when I create the array and copy the array.

The biggest question in the parallel implementation is where to put the cilk_spawn and cilk_sync. The cilk_spawn can be placed in front of the first quicksort call, in front of the second quicksort call, or both. The cilk_sync is called after both of the recursive quicksort calls. I ended up using the cilk_spawn in front of both recursive calls, I determined that it resulted in the fastest times. The parallel code ended up looking like so:

```
parallel_quicksort(array, low, high)
 if (low < high)
  int pi = partition(arr, low, high);

  cilk_spawn parallel_quicksort(low to p)
  cilk_spawn parallel_quicksort(p to high)
  cilk_sync;
```

E. Evaluation

To determine the execution time of the implementation, I ended up using the high_resolution_clock from the chrono library. I originally utilized the clock function from the ctime library but I realized that when using multi-threading the CPU clock numbers are changed and are not reliable for a good reading. The time function is not precise enough.

IV. Results

When n is low, 1000, in all cases the sequential version is quicker and then in the higher n trials the parallel version is better. The tables for both versions are in figure 2 and in figure 4. As you can see in figure 3 the sequential trials resulted in very consistent results. As for the parallel trial you can see in figure 5 the results are inconsistent and reflect the big O evaluations better. As many threads may be called when the trial is less then optimal.

The average speedup is greater than 1 for n values greater then 10,000. The greatest speedup in all the trials is 802% at n of 10,000,000. Shown in figures 6 and 7. In figure 7 you can see that the average and max speed up are relative and the average time taken is very apparent after 1,000,000.

V. Discussion and Further Work

As seen through this report and what I have seen during this project. Many current algorithms that are implemented serially can benefit greatly from parallelization. That is if the number of evaluations reach a certain point. Not only does

Sequential Trials					
Trials\n	1000	10000	100000	1000000	10000000
S1	0.00028	0.00369	0.02421	0.25226	2.84764
S2	0.00033	0.00428	0.02291	0.27799	3.04926
S3	0.00025	0.00335	0.02665	0.24928	2.85099
S4	0.00026	0.00317	0.02127	0.24139	2.87480
S5	0.00028	0.00314	0.02460	0.23852	2.82793

Fig. 2: Table of various trials for the sequential implementation.

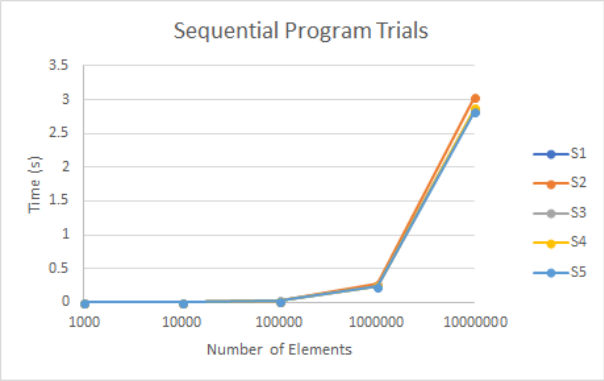

Fig. 3: Graph of various trials for the sequential implementation.

Parallel Trials					
Trials\n	1000	10000	100000	1000000	10000000
P1	0.00038	0.00175	0.03046	0.27483	0.45342
P2	0.00052	0.00715	0.07253	0.19196	0.49362
P3	0.00056	0.00359	0.01752	0.14670	0.46338
P4	0.00031	0.00144	0.00886	0.17614	0.35842
P5	0.00039	0.00126	0.00717	0.16486	0.45892

Fig. 4: Table of various trials for the parallel implementation.

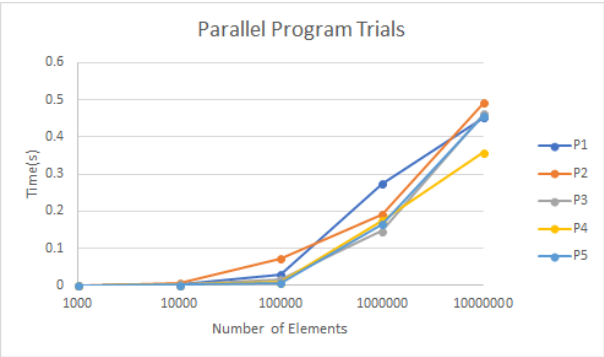

Fig. 5: Graph of various trials for the parallel implementation.

Average Trials and Speedup					
	1000	10000	100000	1000000	10000000
SA	0.00028	0.00353	0.06517	0.25189	2.89012
SP	0.00043	0.00304	0.02731	0.19090	0.44555
Average Speedup	67%	167%	226%	138%	656%
Max Speedup	84%	249%	343%	170%	802%

Fig. 6: Table of average times for both implementations and max and average speedup.

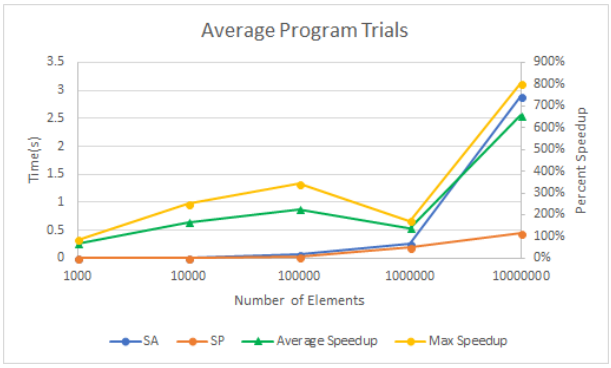

Fig. 7: Graph of average times for both implementations and max and average speedup.

the number of evaluations matter in the scope of execution time but so does the implementation of the parallel libraries.

In this project, I experimented with three locations for the cilk_spawn call. In front of the first quicksort call, in front of the second quicksort call, or both. In my experiments I found that the cilk_spawn in front of the second quicksort resulted in times much much greater than the any other location and even the sequential evaluation.

One of the interesting things that I noticed in this experiment is when the parallel functions are called in a row the latter functions get a boost in execution times. I assume the boost is caused by the generation of the threads in the previous function call and the threads had yet to be kill before they are recalled.

Finally, there are many other methods of optimization to make the program run faster. If you have noticed I did not compile the code with any optimization features, as I wanted to focus on the parallelization speedup. If you include the -O3 optimization you will get a speedup across the board. Furthermore, my implementation of the quicksort algorithm may not be the best. More trials can be done to evaluate the execution times when the first element is picked as the pivot or a random element is selected.

VI. Get Code

You can download the code at :
github.com/vle1054/parallel_computing_final.git