Kennesaw State University

College of Computing and Software Engineering

DEPARTMENT OF COMPUTER SCIENCE

CS4504: Parallel & Distributed Computing

(Section 01) Homework – Spring 2022

Sorting Algorithms: Serial vs. Parallel

Bryson Phillip

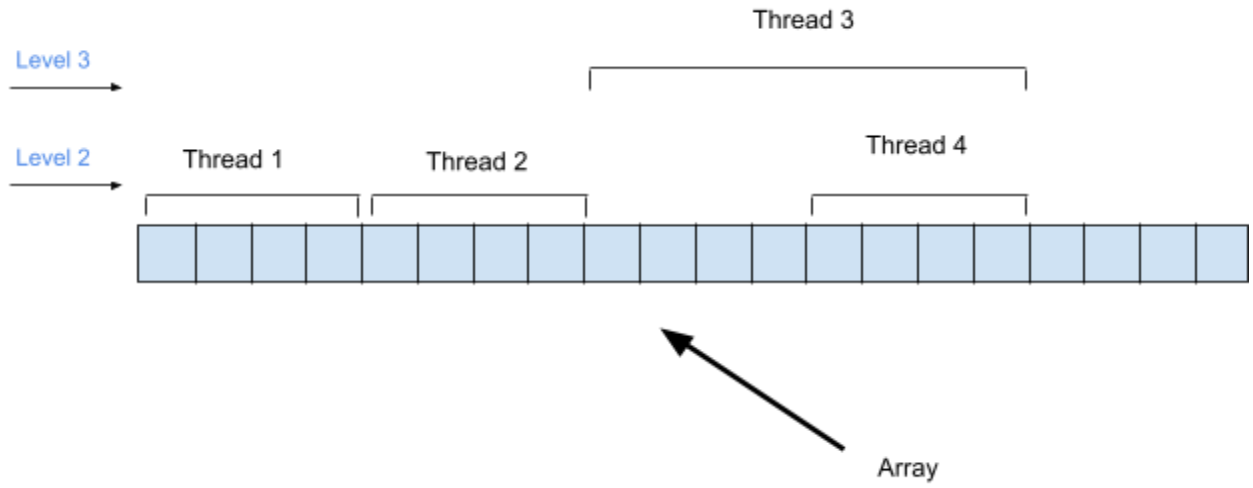
Date: 5/2/2022

Abstract and Introduction

The purpose of this project is to repurpose the skills and concepts used in earlier projects to design an array sorting algorithm. This algorithm should allow for any number of threads operating on arrays of variable sizes and should abstract the interaction of cores (threads) for an ever increasing number of threads. This program will also implement a server-client interaction, with the client establishing a connection with the server and sending the contents of the array, which are randomized. The server will listen for the contents, sort them using threads that are spawned from its main function and return the sorted array back to the client. The objective is to see how aspects of speed and efficiency change as thread pools and array sizes change.

Design Approach

The program is designed to use data parralellism, as opposed to task parallelism. Each thread operates the same as any other thread but will process different, similarly structured, sets of data from within the array. Each thread has a unique id that allows it to be identified when assigning a portion of the array to work on. The algorithm chosen to sort the array requires multiple iterations of sorting over the entire array, and thus threads must, at times, wait for other threads operating within an earlier iteration to finish. However, for threads on parallel portions of the array for the same level, or for non overlapping portions on different levels, process interleaving will be exploited as much as possible.



This illustration shows how data is allocated. Threads 1, 2, and 4 may operate in parallel because they are on the same level and there are no pending lower level processes that use overlapping sections of data. Thread three must wait for thread 4 to be assigned to a higher level, at which point it may begin, even if 1 and 2 exist at a lower level.

Methodology

My program is an implementation of merge sort. Merge sort operates by dividing the array into subsections that are half of the size of the parent until each section is only two indices wide. From there the algorithm travels back up using the sort. In each section there are two pointers. One point points to the beginning of the section, and the section pointer points to the index after the half-way point. The code successively checks which index points to the lower value and shifts the corresponding pointer to the right until it reaches the end of its domain, This results in a new array that combines two sections that had, individually, been sorted by previous iterations. Subsections of the starting array may be processed in series or parallel, but in theory parallelizing should result in speedups if done correctly. Once two arrays are done sorting adjacent subsections, an idle thread can be assigned to sort both of them together on a higher level. This process continues until one thread sorts the span of the starting array. Any idle thread may be assigned to a currently operable section of code. So, the more threads, the greater the portion of the array that can be sorted at once. This is bottlenecked by the requirement that previous subsections must be sorted first. At some point adding more threads will not be helpful as most of them will be idling.

Implementation

The server and client classes operate independently. The server class creates a server socket which listens for the clients connection. The first two sends from the client contain the size of the array and the number of threads. A for-loop then iterates on both sides to send each array index from the client side to the server side. For each pThread created, the run() function calls “set()” which looks at the last assigned thread and

assigns the caller to the next available spot. This is done synchronously to avoid race conditions where more than one thread is assigned to overlapping spots. At each level, spots are assigned from right to left across the array. The range of values to be sorted by a given thread is calculated as the (start-1) + 2^(level) where level is increased after threads reach the end of the array. (start-1) marks where the range of the previously assigned thread ends. The span increases as 2, 4, 8, 16…

Sorting is done by creating another temporary array ‘temp’ to store the values selected by each pointer in the subarray. Once the data is sorted, the values of the temp overwrite the original array values at the appropriate start and end points.

```
// implementation of  merge sort
int []temp = new int[ (int)Math.pow(2,level) ];
int index1 = start;
int index2 = start + ( (int)Math.pow(2,level) / 2);
int temp1;
int temp2;
for (int i = 0; i < temp.length; i++) {
    if ((index2-start < temp.length) && (index2 < arr.length)) {
        temp2 = arr[index2];
    }
    else{temp2 = 100;}

    if ((index1-start < (int)Math.pow(2,level) / 2) && (index1 < arr.length)) {
        temp1 = arr[index1];
    }
    else{temp1 = 100;}

    if (temp2 < temp1) {
        temp[i] = temp2;
        index2++;
    } else {
        temp[i] = temp1;
        index1++;
    }
```

```
    }
    for (int i = 0; i < temp.length; i++){
        try{
            arr[start + i] = temp[i];
        }
        catch (Exception io){}
    }
```

This code shows the execution of merge sort for a sub array and subsequent reintegration.

```
static synchronized void set(pThread self, int arrlength){
    // checks the position and level of the previously assigned thread and assigns
    // an idle calling thread to the next position
    if (!first) {
        int temp1 = glob_level;
        int temp2 = glob_start + (int)Math.pow(2,glob_level);
        if (temp2 >= arrlength) {
            temp1++;
            temp2 = 0;
        }

        self.level = temp1;
        self.start = temp2;
        glob_level = temp1;
        glob_start = temp2;
    }
    else{
        first = false;

    }
}
```

This code shows how calling threads are assigned sections of the array to sort at the appropriate level. The function will ‘wrap-around’ to the start of the array at the next level if the next logical position is out of range.

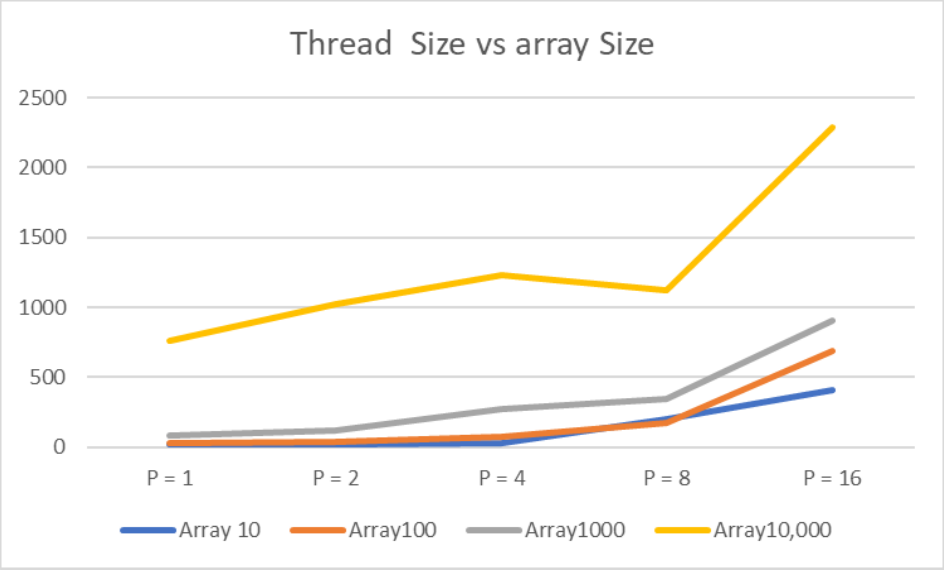
Results

```
Type true to run in multi-thread mode, otherwise type false:
false
Array size:
10
[19, 19, 0, 34, 76, 18, 58, 51, 31, 28]
[0, 18, 19, 19, 28, 31, 34, 51, 58, 76]

Process finished with exit code 0
```

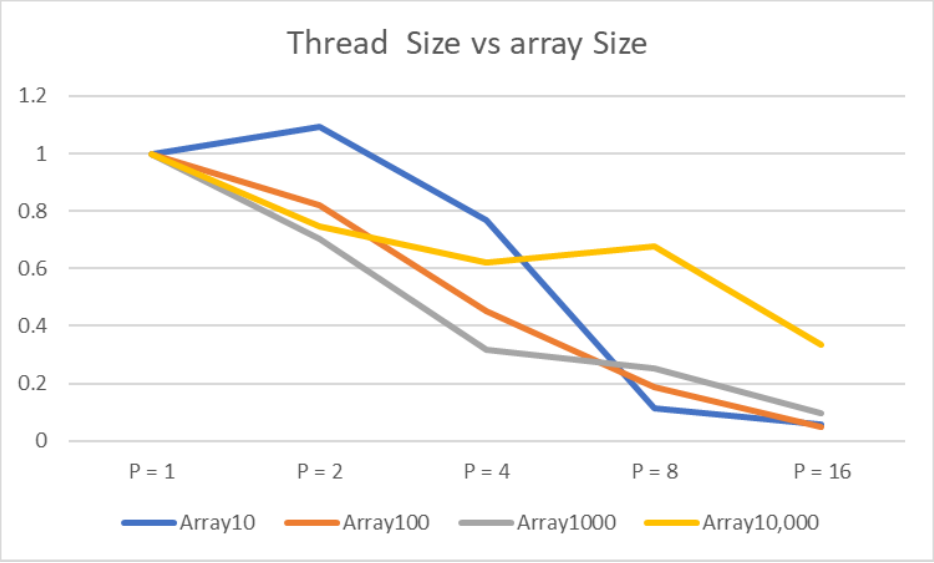
Time in milliseconds

No of Threads	10	100	1000	10,000
P = 1	23	32	86	761
P = 2	21	39	122	1021
P = 4	30	71	272	1227
P = 8	205	169	344	1123
P = 16	407	684	905	2285



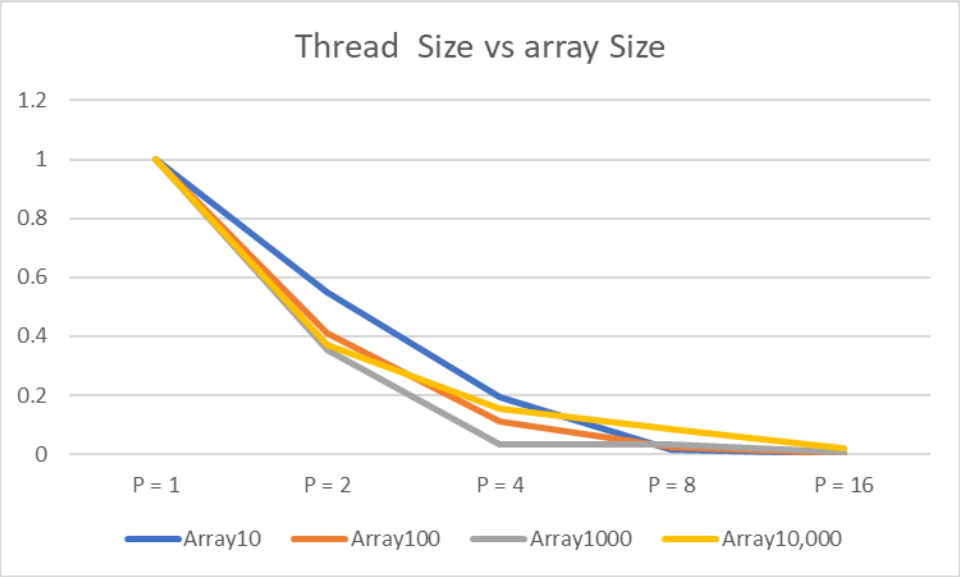
Speedup

No of Threads	10	100	1000	10,000
P = 1	1	1	1	1
P = 2	1.095	0.821	0.705	0.745
P = 4	0.767	0.451	0.316	0.620
P = 8	0.112	0.189	0.250	0.678
P = 16	0.057	0.047	0.095	0.333



Efficiency

No of Threads	10	100	1000	10,000
P = 1	1	1	1	1
P = 2	0.548	0.411	0.353	0.373
P = 4	0.192	0.113	0.034	0.155
P = 8	0.014	0.024	0.031	0.085
P = 16	0.004	0.003	0.006	0.021



Results and Conclusion:

The results of these tests would seem to be counterintuitive at first. One would expect that speedup would increase with the number of cores because multiple cores can do work at the same time. However, this can only work if it is running on a computer that has multiple processors. In my case, my computer only has one core and each operation for each thread is being interleaved. When considering the context switch time between threads, the time can only increase. Every operation that would be done in series without threading is also done in series with threading, just with more interleaving. This would not be the case if my computer had multiple processors. In addition to the negative impact of context switch time, there would also be multiple p threads checking if the assigned spot may be operated on, this uses up cpu time that wouldn't be used with just one thread.

Bottlenecking can also be seen for the low array size and high thread size combination. When there are more threads operating on an array than can be allowed by the size,

some idle arrays will use resources that don’t ultimately go towards sorting. For example, idle arrays will still be given access to the set() function which may block other more active threads from being reassigned. Their checking loops will also eat up resources. We can see that speedup generally increases as array size increases because there is less competition between threads.