CS 426/525
Fall 2022
Project 2
Due 25/11/2022, 23:59

1. Introduction

This project will focus on the Quicksort algorithm and its parallel implementation using MPI. You are going to use C or C++ programming languages.

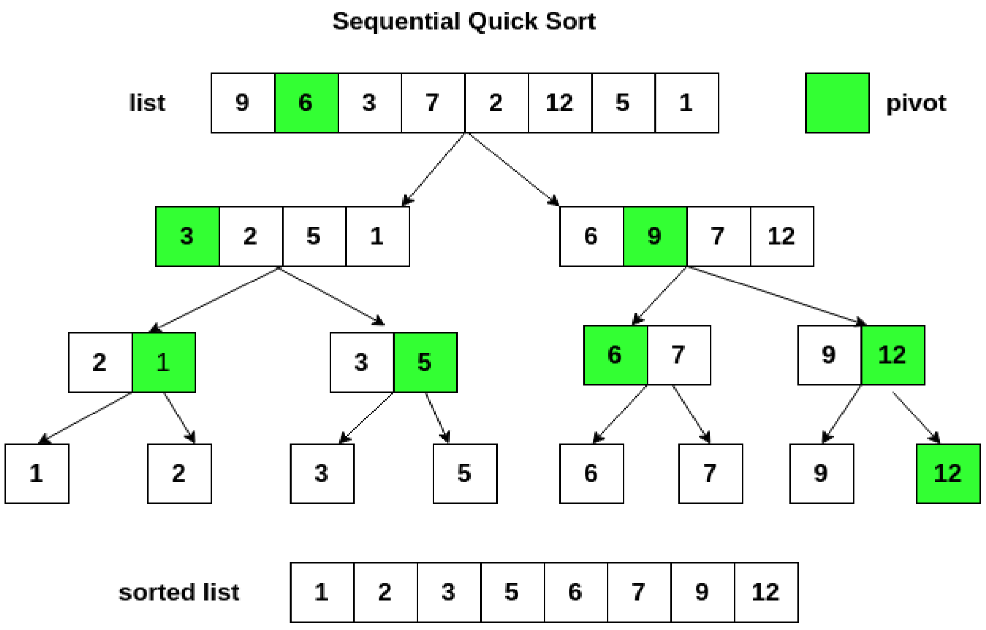

Figure 1: *Steps of the Quicksort algorithm in sequential execution.*

2. Serial Quicksort

Quicksort is one of the most common sorting algorithms for sequential computers because of its simplicity, low overhead, and optimal average time complexity.

Quicksort is a divide-and-conquer algorithm that sorts a sequence by recursively dividing it into smaller subsequences. Assume that the n-element sequence to be sorted is stored in the array **A[1...n]**. Quicksort consists of two steps: divide and conquer. During the divide step, a sequence **A[q...r]** is partitioned (rearranged) into two nonempty subsequences **A[q...s]** and **A[s + 1...r]**, such that each element of the first subsequence is smaller than or equal to each element of the second subsequence. During the conquer step, the subsequences are sorted by recursively applying quicksort. Since the subsequences **A[q...s]** and **A[s + 1...r]** are sorted, and the first subsequence has smaller elements than the second, the entire sequence is sorted.

Implement the Quicksort algorithm with either C or C++ language. *This part will allow you to remember and rethink the details of Quicksort*. Your program will take input as an array of positive integers, stored in an ASCII file with one integer per line, and writes the sorted array to an output file.

```
# cat input
8
2
20
5
35
63
98
41
# ./qsort input.txt output.txt
2
5
8
20
35
41
63
98
```

3. Parallelizing Quicksort

There are several ways to parallelize Quicksort. What we will focus on in this project is, implementing the Quicksort on a Hypercube. Let the communication network topology be a **d**-dimensional hypercube (i.e., the number of processors is equal to **p=2^{d}**). A hypercube of size **n** consists of **n** processors indexed by the integers {0,1, . . . , n −1}, where n > 0 is an integral power of 2. Processors A and B are connected if and only if their unique **log2 n-bit** strings differ in exactly one position. We define processors A and B as partner processors.

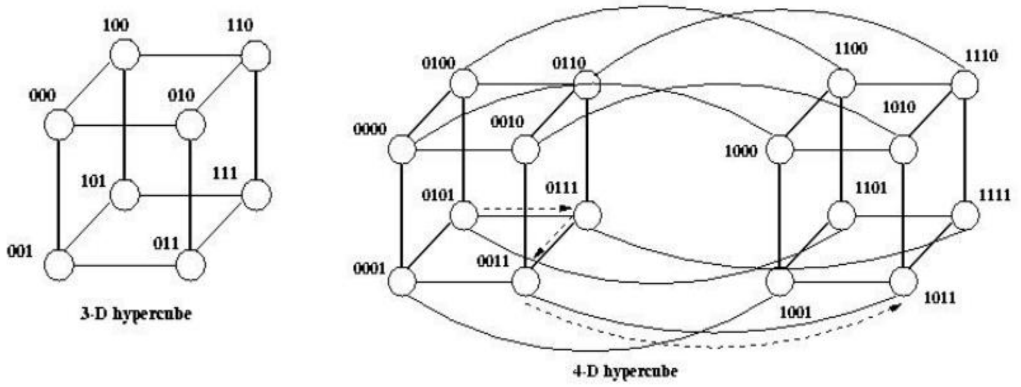

Figure 2: *Processor distrubiton on a 3-dimensional and 4-dimensional hypercube.*

There are different ways to select pivot in parallel Quicksort. A variant is a random processor choosing a pivot, the median of its local list, and broadcasts the pivot to other processors. We will examine the steps of the algorithm in a case in which we have a hypercube of dimension **d = 2** and a list of size 32.

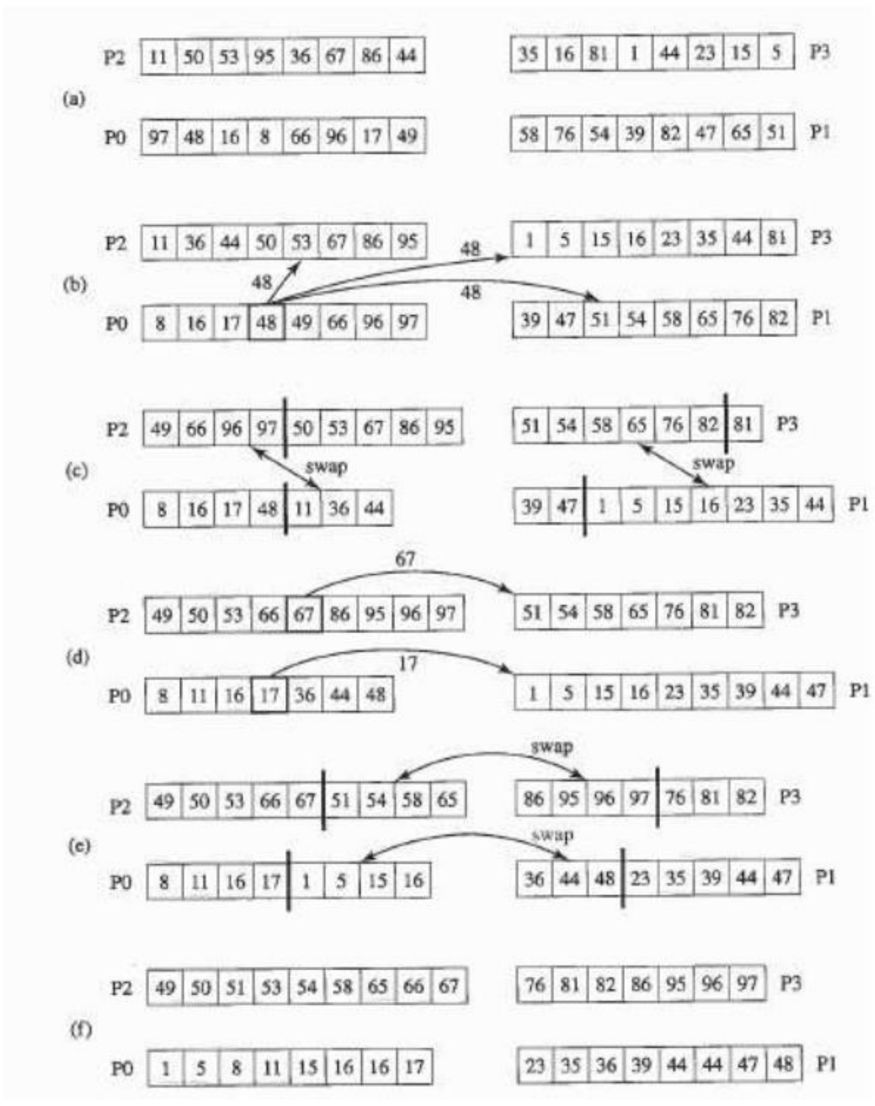

***Figure 3**: Steps of parallel quicksort on a hypercube of dimension 2 (original example from: Michael J. Quinn, Parallel Programming in C with MPI and OpenMP).*

You can see in the above figure that processors P0-P2 and P1-P3 are partners and exchange their upper and lower sublists with each other in step c. The hypercube split into subcubes in part d. Notice that two separate communication networks spawn here. Therefore, a random processor from each communication network will calculate and broadcast its median as a pivot **in its communication network**. You are going to spawn communication networks using the **MPI_Comm_Split** function.

We will modify pivot selection to preserve load balance among processors (the Parallel Quicksort algorithm is likely to do a poor job in load balancing). **Instead of a random processor calculating the median of its list, as in Figure 3, every processor will calculate its local median, and the pivot will be selected as the median of medians**.

You will use collective communication functions (***MPI_Allgather*** as an example) to ensure all processors will do local sorting according to the same pivot.

Your program will accept an array of integers stored in an ASCII file with one integer per line, as in the first part. Steps of the algorithm are as follows:

1. The master reads the file, and integers are distributed to all processors.
2. Each processor sorts its local list using sequential Quicksort.
3. Each processor finds the median of its local list.
4. Median of all medians is found and assigned as the pivot.
5. Each processor splits its local list into upper and lower sublists with respect to the pivot value.
6. The current hypercube is split into two sub-cubes. Each processor will communicate with its partner processor of the sub-cubes and exchange either the lower or upper list of elements. Since these lists are already sorted, they can be merged into a sorted list.
7. Recursively repeat steps 2-6 on the sub-cube until the list is completely sorted. When the sub-cube only contains one processor, the list should be in completely sorted order.
8. Sorted local lists are merged into a single list. This operation could be done by either collective communications or processors sending their local lists to the master, which will merge them with its own.

At the end of the program execution, each processor should write out their sorted local lists as a text file in the same format as the input. Also, a complete list formed by merging sorted local lists should be written as a text file.

Your program will run in the following format:

```
# ./hyperqsort INPUT_FILENAME.txt -np OUTPUT_FILENAME.txt
```

For example, a configuration of 2-D hypercube (4 processors) should be able to run as;

```
# ./hyperqsort input.txt 4 sorted.txt
```

Generated files should be;

```
# output0.txt output1.txt output2.txt output3.txt sorted.txt
```

4. Assumptions and Hints

- There will be log_2P recursions, where P = *number of processors*.
- Assume number of processors is a power of 2.
- Assume size of the input array is divisible by the number of processors with no remainder.
- The expected number of times a value is passed from one process to another is $(log_2P)/2$.
- You can use **bitwise XOR operatör (^)** to decide which processors will be partners and will exchange their upper and lower sublists. For example, in a hypercube of dimension 3; *P0-P4, P1-P5, P2-P6, P3-P7* will be partners.

5. Grading

- Sequential Quicksort: 10
- Parallel Quicksort: 70
- Report: 20

6. Submission

Submit a single zip file to Moodle (**yourname_lastname_p2.zip**) that includes:

- Your implementation with source files and necessary inputs for the following;
  - qsort.cpp
  - hyperqsort.cpp

- A Makefile that generates the following 2 executables;
  - qsort
  - hyperqsort

- Reports **should be in .pdf format**, submissions with wrong format will get 0. Your report should include (4 pages at most):
  - Briefly explain your implementations. Plot graph(s) with various thread numbers for small, medium, and large input sizes indicating the performance of your implementation by comparing serial and parallel implementations. What are your observations? Does parallel execution improve the performance of any inputs? If not, what might be the reasons?

Email: utku.gulgec@bilkent.edu.tr

Email subject: CS426_Project2

Zip File name: yourname_lastname_p2.zip (Without this name, will not be evaluated).

No Late Submission Allowed!

7. Useful Links

- https://iq.opengenus.org/parallel-quicksort/
- http://users.atw.hu/parallelcomp/ch09lev1sec4.html
- https://dl.acm.org/doi/pdf/10.1145/130069.130085
- https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Mrunal-Narendra-Inge-Spring-2021.pdf