| CSci 5451, S'20 | Programming Assignment # 2 | Due Date: April 1, 2020 |
|---|---|---|

# Background

This assignment will familiarize you with MPI and some of the basic issues of parallel algorithm design. The main task is to program a parallel version of the quick–sort algorithm. The idea is to think about an implementation of this algorithm on a hypercube. You need not have a hypercube network but all the thinking should be guided by the topology of a hypercube. The problem formulation is as follows. We have $p$ arrays or equal, or roughly equal size, one in each of $p$ processes and want to sort the *union* of these arrays in an ascending order. The goal is to have the final result distributed, so that each array held in a process is sorted and the entries of the array in process $P_i$ are larger than those of the array held by process $P_{i-1}$, for $i = 1, \cdots, p-1$.

# The algorithm

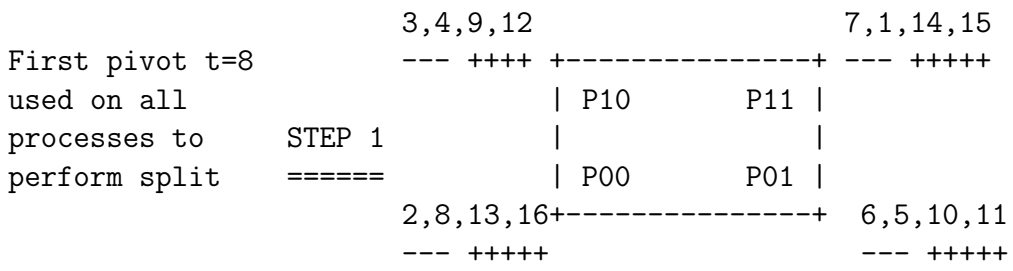
It may help to start with a review of quicksort by taking a look at standard texts (see also mergesort as merge operations are extensively used). In the following $p$ is the number of processes, $q = p/2$. The root process is typically process number zero. Here is a high level description of the algorithm:

1. Data is initially distributed in arrays $A_0, A_1, \cdots, A_{p-1}$ in processes $P_0, P_1, \cdots, P_{p-1}$
2. Each process $P_i$ sorts its own array $A_i$. [You may use the provided mergesort function for this]
3. Process 'root' selects a list of $p-1$ pivots and broadcasts these to all processes. [The function that sets up this list of pivots is provided]
4. Each process $P_i$ splits array $A_i$ into $A_i^{high}$ and $A_i^{low}$ using the pivot $t$ for this step (at the end we will have: [1]: $A_i^{low} \leq t < A_i^{high}$)
5. Processes $P_i$ and $P_{i+q}$ with $i < q$ perform an exchange: $P_i$ sends $A_i^{high}$ to $P_{i+q}$ and $P_{i+q}$ sends $A_{i+q}^{low}$ to $P_i$.
6. Each process now merges its new arrays ($[A_i^{low}, A_{i+q}^{low}]$ in $P_i$ and $[A_i^{high}, A_{i+q}^{high}]$ in $P_{i+q}$) into sorted arrays (using a `merge` function – also provided).
7. Processes $P_0, \cdots, P_{q-1}$ and $P_q, \cdots, P_{p-1}$ now form two groups of processes. Each of the processes in each group holds a sorted array. We now repeat each the above steps recursively on these two groups of processes until the groups contain only one process each.

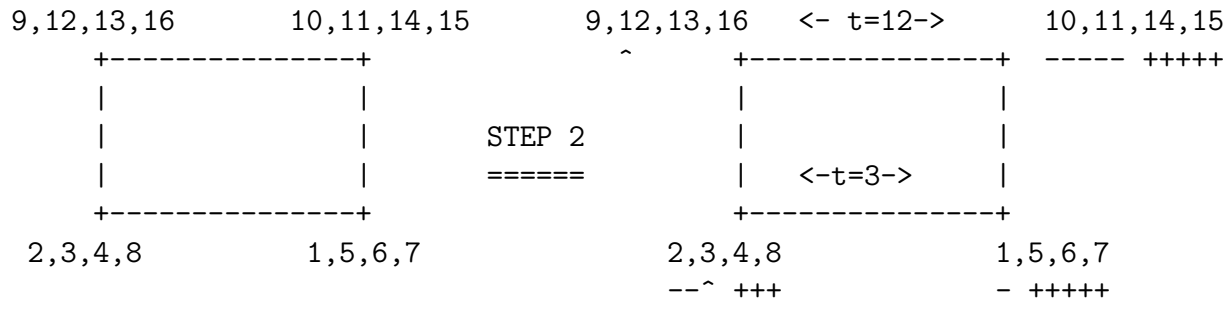
---

[1] Notation: $X < t$ means all values of array $X$ are $< t$.

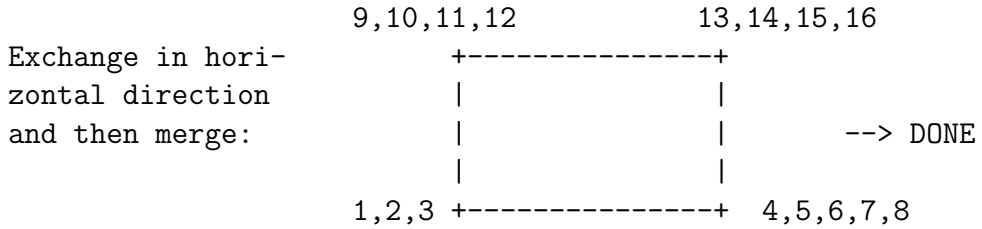
Here is an illustration with $p = 4$. The four arrays located initially located in processors 00, 01, 10, 11 are $A_{00} = [2, 4, 13, 16]$, $A_{01} = [6, 5, 10, 11]$, $A_{10} = [3, 4, 9, 12]$, $A_{11} = [7, 1, 14, 15]$. Assume that the list of $p - 1 = 3$ pivots found in this case are $8, 12, 3$. This list is broadcast to all processors.




Each node splits its array in two parts: larger than the pivot underlined with `+++`) and $\leq$ pivot (underlined with `---`). We then exchange in the vertical direction (direction corresponding to last bit in hypercube) the "++" side of the arrays at bottom with the "—" side of the arrays at the top. Each of the arrays is sorted by a `merge` operation (result: left side of next figure).




We now work in subcudes. Nodes labeled 0x (P00 and P01) use the pivot t=3. Those labeled 1x (P10 and P11) use the pivot $t = 12$




---

A few observations. First although we use recursivity to describe the algorithm, in reality the code which you will write need not (and should not) be recursive. Second, one may ask why the algorithm works. As a result of step 5 (exchange), all processes with high labels will have their entries larger than the pivot $t$ and all those numbered less will have their items not larger than $t$. By recursivity, the sizes of the entries held by processes will evolve like the process numbers. In the end process 0 holds the smallest entries (sorted), process 1 holds the next smallest entries, etc.. Another illustration of this algorithm (with 8 nodes) is provided in pages 7-28 to 7-30 of lecture notes set number 7.

# Implementation

We will deal first with the issue of selecting the pivots. The performance of the algorithm is very sensitive to this selection and if we are not careful, we will end up with arrays of very imbalanced sizes. Ideally, the pivots $t$ that are used in each direction of the cube should be selected dynamically from medians of sorted arrays in subcubes but this is somewhat lengthy to code. Another way to deal with this is to gather some data in the root node and select the pivots ahead of time. To save time you will be provided with a simple code that does this. The root process gets some data, computes good pivots and broadcasts these to all processes. This is greatly facilitated by the fact that the arrays are sorted initially. Therefore, step 3 in the previous description, picks the pivot from an array `list` which is initially broadcast by the root node to all processes.

The main loop of your `HQuicksort` (for Hypercube quicksort) algorithm is along `ncub`, the dimenson of the hypercube ($\log_2(p)$). Determine $ncub$ as log2(p) and exit with error if $p$ is not a power of 2.

At each step of the $j$ loop, processes will form pairs $(P_i, P_k)$. In the illustration given in the previous section these pairs are $(P_i, P_{i+q})$, where $q = p/2$. A low-numbered processes in each pair sends its high part of its array to its partner. It will in turn receive the low-part of the array if its (high-numbered) partner. One of the main issues you will need to resolve is to how to determine these pairs [Hint: This is actually fairly simple. At each step $j$ the pairs are those processes whose $j$-th bits form a zero-one pair and the other bits are identical. Think in terms of powers of 2.]

# Tests

A main program will be provided. Also provided are the routines for selecting the pivots and some additional functions such as `merge`. The goal is to cut down on coding time and allow you to focus on the main function which is the `HQuicksort` function. The main program will generate an equal number of random integers on each process. Your function should sort the aggregate of all the numbers using all processes. The main program gathers these arrays into a global array to check that the sorting worked properly.

You will first test your code with $ntot$ (total size of data) set to 200 in `main.c` (provided) and and run the code with $np = 4$. The purpose of this is to show that your program does indeed work. Do not change the main program for this test.

Once your code has been tested you will need to analyze its performance and see, for a large number of items to sort what happens to the execution time and to the imbalance between array sizes as the number of processes increases. Here you will set the total number of items to sort to be equal to maybe $ntot = 100,000$ and vary the number of processes from 2 to 16.

You can provide a plot (or a table) of the timing achieved [2]. For this test you will need to modify `main.c` to remove the unnecessary print statements, and to add calls to timing functions (use MPI_Wtime()).

[2] It is not clear how this will work given the number of students working on the project at the same time. But your analysis of the results you get should be on what you see

# What to submit and grading criteria

## Submit:

(1) All source codes. These should be submitted by the online "submit" program. There should be two main programs. The one provided [unchanged]. Call this `main.c`. Then you will also need to provide another main with stats [timing and standard deviation for sizes.] Call this `mainStats.c`. You may then provide makefiles for both executables. main.ex and mainStats.ex. A README file explaining how to make the executables and run them may help but is not required if the makefile is self-explanatory.
(2) Provide a file called `Report` (or `Report.pdf`). This can be a PDF file with plots. It can also be a plain text file with tables. Here are some of the items you need to comment on.

1. Any specific comments you have on your implementation. For example did you use any `MPI_Barrier` commands? If so why were they needed? Did you have to use any communication commands other than sends and receives in your function?

2. Provide an analysis for the execution time in the best case scenario when sets are always perfectly balanced. The analysis model should count the number of comparisons as well as account for communication operations. What does the model tell you (consider situations of large $n$ and small $n$ relative to processor numbers).

3. Comments on the statistics you see. Timing, performance in terms of load balancing, etc.

4. Finally, any comments on what you could do to improve performance. In particular, how can you improve the pivot selection strategy if you had to write a real 'production' code.

## Grading:

1. **15** % Style and documentation.

2. **30** % Overall correctness of your sorting routines

3. **30** % Correctness of the specific approach [i.e., how does it conform to what is being asked.]

4. **25** % Quality of your report: your comments, your suggestions for improvements on pivots, etc..