**PARALLEL SYSTEMS: BITONIC SORT USING MPI**

In this assignment, the goal was to succeed to sort a large array of integers contained in different processes using a MPI (Message Passing Interface). To solve this problem, we have to use the bitonic sort, a sort designed to be efficient on parallel structures. We will first see what the program is supposed to do and then discuss my implementation.

The program is supposed to take 2 arguments: the number of process to be created and the number of integers contained in each process, both powers of two. Then, through communication between the processes, the array must be sorted. In the end, the process 0 should have the 2^q lower integers, process 1 the 2^q higher following, etc… until the end.

To do this sorting, I used the imperative version of bitonic sort. The one provided was to sort a simple array, so I had to modify it to sort an array of arrays. The only major change was to delete the third loop which contained the index of the array, because now the index is the index of the process processing this function. Also I had to add an “else” block for the (ij>taskid) condition because the paired process have to execute a procedure as well.

In the loop, ij is the process paired to the current process (base on j and the current process), and k is the variable used to determine the direction of the sort.

Inside this function, I have to other functions: exchangeLow and exchangeHigh, which take the ID of the paired process as arguments. If one is called, the paired process always calls the other.

The process calling exchangeLow, after execution, will contain the lower part of the 2 arrays and the one calling exchangeHigh will contain the highest part of the 2 arrays. Those functions use the following property:

If a sequence sorted ascending is compared element by element to a sequence sorted descending and the elements are switched one by one if process1[i]>process2[i], we obtain 2 bitonic sequences with all elements of the first being inferior than all elements of the second. So when we sort ascending and descending those sequences using qsort, we obtain a new bitonic sequence.

This is the case if process1 calls exchangeLow, meaning that process1 will contain in the end the lower elements.

The algorithm is really close to the serial one, because the order of the comparisons between the arrays follows the same pattern as the serial comparisons, the only differences are in the functions called.

The exchangeLow procedure works as follow:

First, the array of the process is locally sorted, depending on the paired process, so as we always have an ascending/descending structure to compare every time.

Then, the process sends the upper part of his array to the paired process, and receives the lower part of the paired process in a buffer. Every buffer’s element that is lower than the array’s element is swapped from index 0 to N/2, and then we send back the buffer to the paired process. To finish, we receive the updated upper half of the array we send and paste it on the upper half of the local array.

The exchangeHigh procedure is quite similar, but the comparisons are on the upper half of the array (N/2 to N-1) and this array only keeps higher values.

When the principal loop is over, we do a last quick sort of all the arrays in ascending order to obtain the final result. I also implemented a test procedure to check if the array is correctly sorted, using also MPI.

Please find attached to source code.

In the end the program is running, maybe some optimization is possible on the comparing loops, to save some cycles. There is I guess a lot of other ways to do such a sort using MPI, but this is the most convenient way in my opinion.

**RESULTS**

Now, let’s take a look at some measurements of the tests I ran on Hellas grid, for q from 16 to 20 (the data size) and p from 1 to 7. I will just show some graphs that are representative of my results so as not have to show too much redundant information.

As we can see from those graphs, the time of computation grows as the size of the array grows, which is perfectly normal as the quicksort () function will take much longer on bigger arrays.

Regarding the time being bigger for the same q for different number of processes, let’s take a look at this graph:

If there was no message passing between the processes, this curve would be flat, because each core would spend exactly the same time to do the local sort. But as there is more and more process to communicate with, the time of the computation gets longer because of the time of the communications and also because of the local sorts. There are indeed more of them the more process there is, as the bitonic algorithm requires.

The improvement compared to the serial code is really significant. For 2 process with q = 20, the MPI program takes 0.5 seconds to sort the array of 2^21, and the bitonic serial code takes 6.5 seconds, and the gap gets bigger and bigger : for 64 process (so an array of 2^26 elements), the serial time is 280 seconds compared to 5,65 seconds for the MPI program ! (I wasn’t able to run for 128 process on hellas grid)

If we change the bitonic serial code to a quicksort, the gap is smaller, but still widely significant : for an array of 2^64 elements, it takes 64 seconds for quicksort compared to the 5,65 seconds seen before.

Maybe my code is not perfectly optimized, and the time for a same q and a larger number of process could be diminished, maybe by using non-blocking send and receive somehow, or break the comparative loop under some conditions that don’t affect the sort.

This assignment proved anyway that using a computational grid provides really improved results using MPI compared to a serial code, with a large number of elements, and should be used in case of a need of a computation of very large data.