Fabien Labarbe-Desvignes

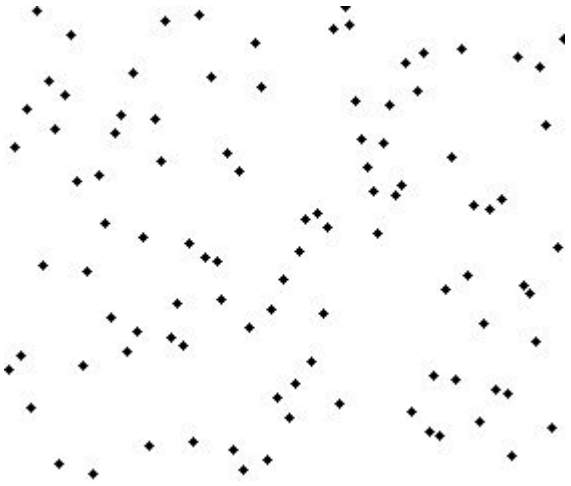
Parallel odd even sort transposition

Summary

- Overview of the project
- Build & usage
- Solution design and architecture
- Obtained results and analysis (MPI)
- Obtained results and analysis (Pthread)
- Performance analysis
- Learnings & conclusion

Overview of the project

In this work, I had to write a parallel odd-even transposition sort by using MPI.



An Odd-Even Sort is a simple sorting algorithm, which is developed for use on parallel processors with local interconnection. It works by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order the elements are switched. The next step repeats this for even/odd indexed pairs. Then it alternates between odd/even and even/odd steps until the list is sorted.

The odd even sort is fitted for parallel computing since you can swap pairs of the array independently from each other.

Build & usage

Requirements:

cmake (for compilation), pthread, mpi

How to build the parallel / sequencial version?

1 go to 119100002/parallel or 119100002/sequential

> cd 119100002/parallel

2: create a build folder, go in it,compile the cmake, make and come back

> mkdir build && cd build && cmake .. && make && cd ..

How to run the program manually

./even-odd-sort VECTOR_SIZE SEED MOD [NB_THREAD]

VECTOR_SIZE: size of the vector containing the random numbers that will be generated

SEED: seed on which the random numbers will be generated

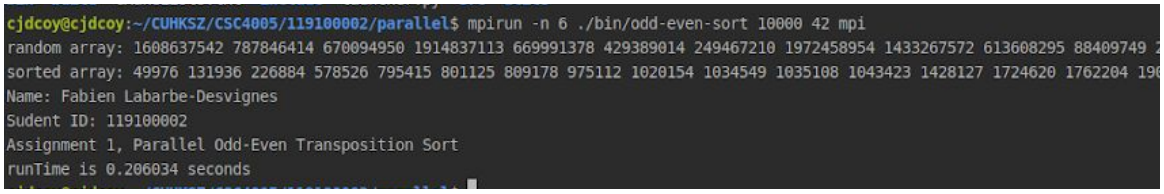
MOD: "mpi" or "pthread"

NB_THREAD: unnecessary for mpi, required for pthread (number of threads that will be used

Examples with parallelization **(array size 10 000, seed 42, processes 6)**

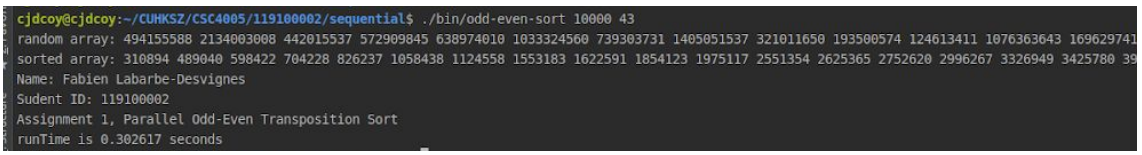
MPI: mpirun -n 6 ./bin/odd-even-sort 10000 42 mpi

PTHEAD: ./bin/odd-even-sort 10000 42 pthread 6



Example with sequential **(array size 10 000, seed 42)**

./bin/odd-even-sort 10000 42



Run the program with python3 for automatic graphics

You can use the python3 script I made for benchmarking if you want.

requirement: matplotlib (python3)

command line to run the python3 launcher:

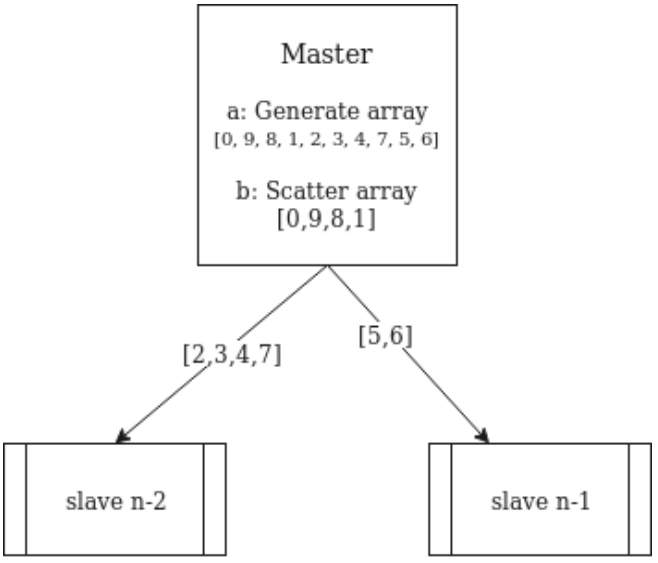
> python3 launcher.py

Algorithm design

My algorithm can be divided in 3 main phases:

1. Pre-processing
    a. Array generation
    b. Array scattering
2. Processing
    a. Inner-process swap (odd)
    b. Inter-process swap
    c. Inner-process swap (even)
    d. Stop / continue signal
3. Post-processing
    a. Array gathering
    b. Array displaying

I'll now introduce each phase more in detail so you can have a full understanding of my solution.



Phase 1:

a: Master generate a random array

b: Master scatter this array in n parts (n=number of process). Master keeps the first part of the array since he'll swap altogether with the slaves.

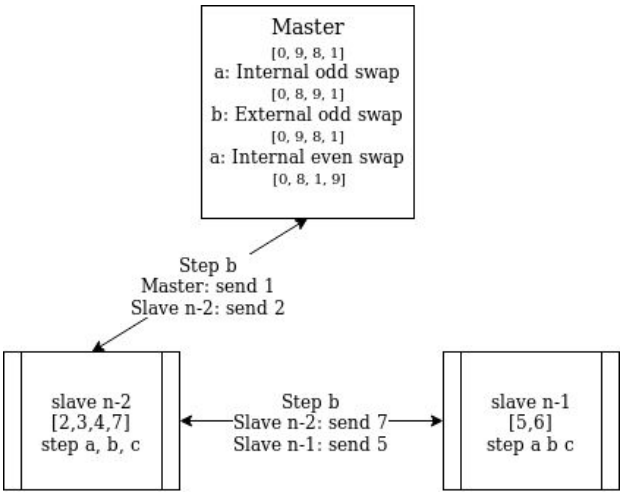
note: Every array beside the last one has a pair number of elements. This allows us to have less computation during the inter-process swap, the drawback of this method is that the last one have either less or more swap than the others to do but on large array this is negligible.

Phase 2:

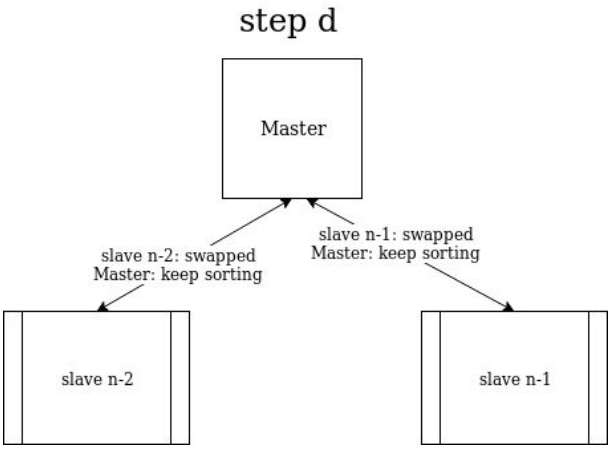
a: Each process swap all its odd pair once (if needed)

b: Each process send the last number of its array to the next process and its first array number to the previous process.
Each process swap depending on its own values and on the one received.

c: Each processes swap all its even pair once (if needed)



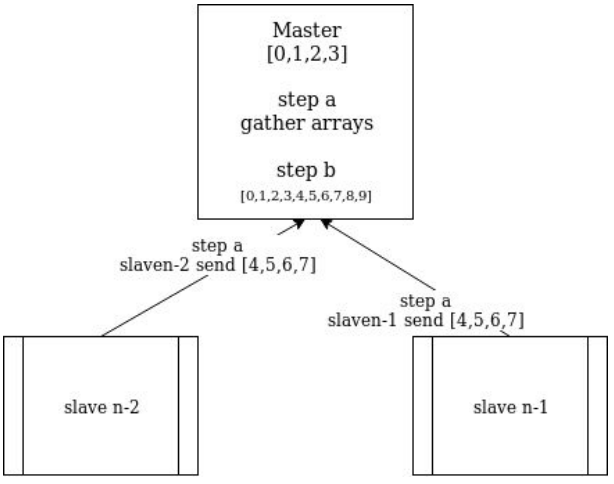
d: Each slave notify master if he performed a swap. If a swap was performed, the Master broadcast false, otherwise it broadcast true and all the processes exit the odd-even sort loop.



Phase 3:

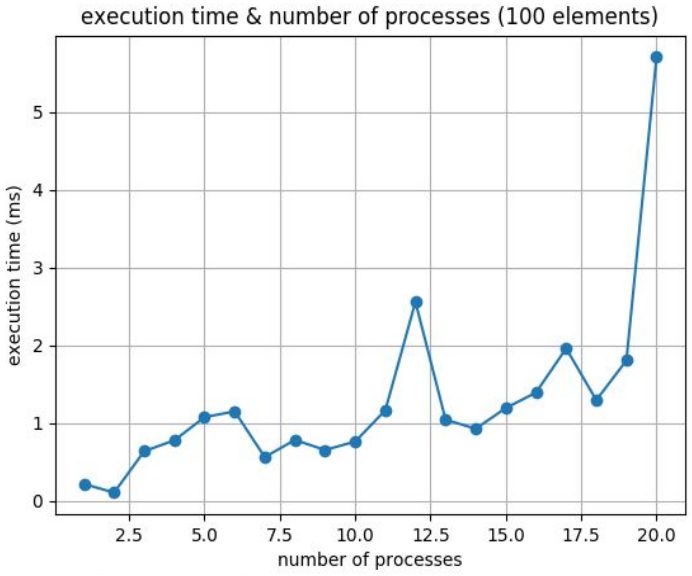
a: After exiting the odd-even sort loop each slave will gather its sorted array to the master and then exit.

b: The master can do whatever he wants with the sorted array

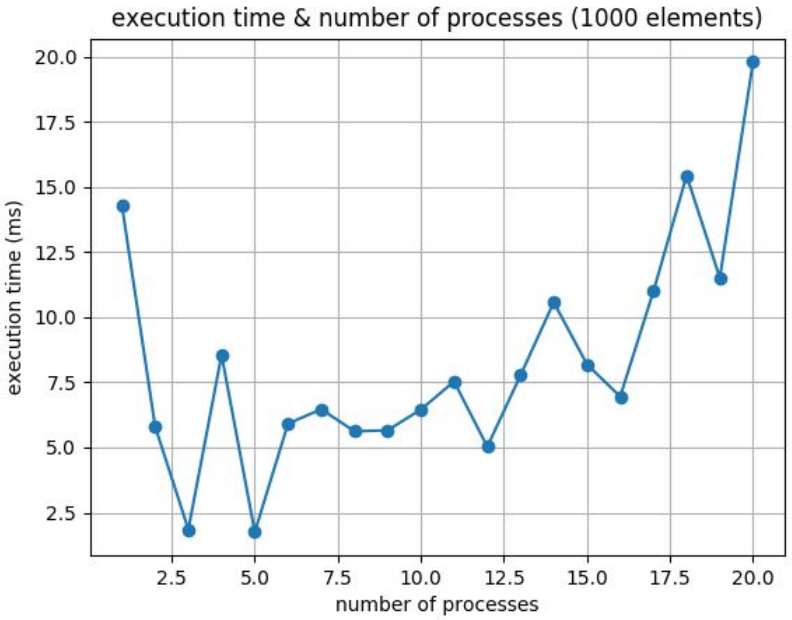


Obtained results (MPI)

Note: The following computation have been performed on a i7 9700K (12 cores)

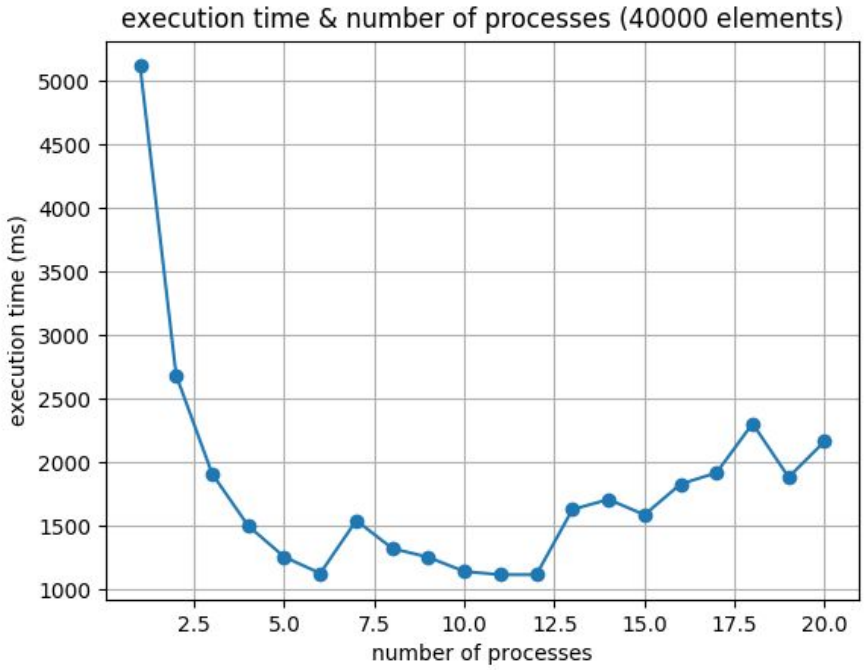


On this figure we can see that the the more processes we have the more the longer the execution time is. This is mostly due to the number of communications between each process because communications are slower than swaps.



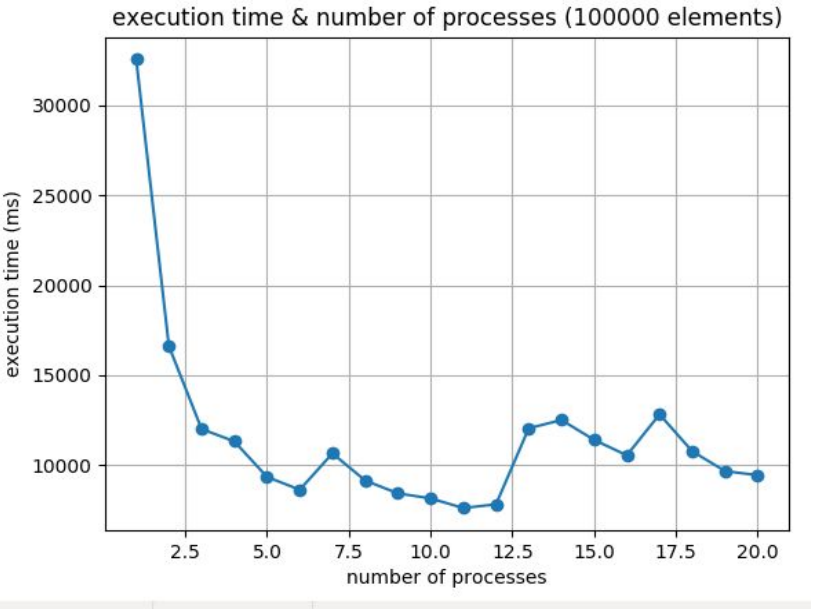
On this second figure we can see the same kind of phenomenon happening but with an improvement: having multiple processes is faster than having just one as long as you're not above 16 processes, after that the number of communication is too important compared to the number of inner swaps.

From this figure we can also see that the more processes we add, the slower the computation gets, for the same reasons as the previous case.



On this figure we can see that the speed is slowly getting better and better as we add more processes until 12. At 12 we ‘re getting slowed down because of the hardware limitations: after 12 processes every core reach 100% utilization which slows the computations.

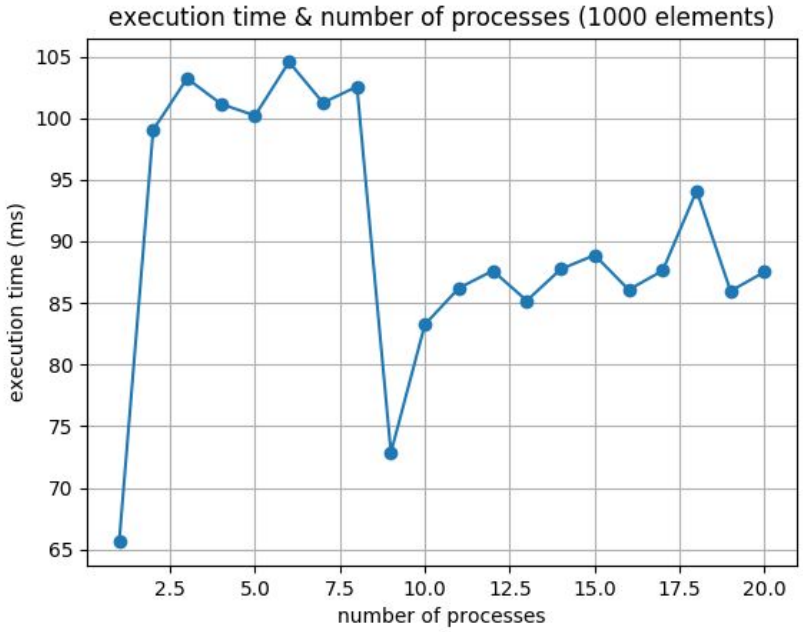
We can also note that the best performance is reach at 12, when all of the computer’s core are used by one process.



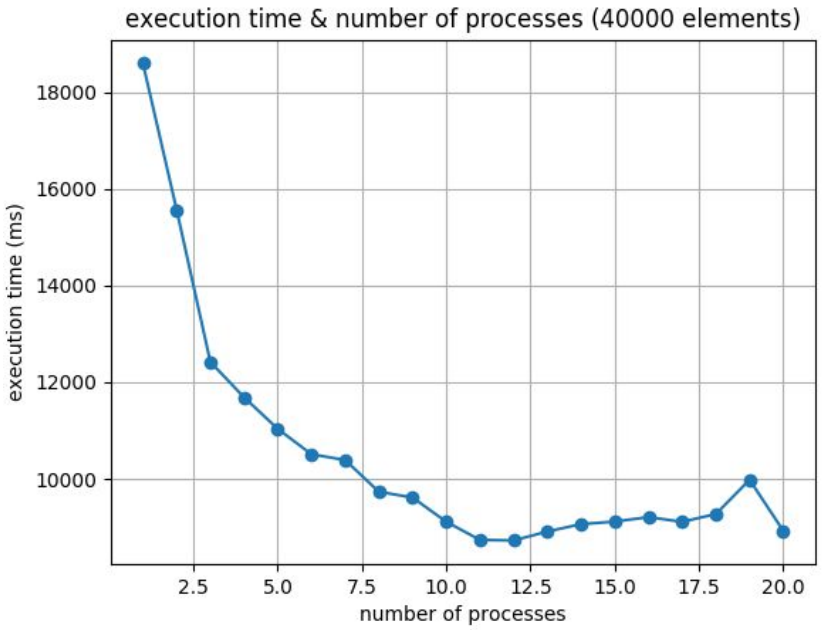
This last figure allows me to confirm that 12 process is the best for my actual configuration but it also show us that the bigger the array the more we can see accurately the difference of performance due to the communication between the processes.
Indeed, figure is pretty much the same as the previous one but we can see more precisely the difference of performance because there are more a lot more swaps than communications.

Obtained results (Pthread)

The same configuration is used for pthread and MPI. Since the work is not about pthread I'll sweet the details, just note that the scattering is the same as in MPI but there's no communication between the processes. The stop condition is different, each thread update an atomic bool on swap and an atomic int to synchronise the threads. I'm using atomic variables to avoid any concurrency problem though it slows down the processing more and more depending on the number of processes.



On this figure we can see that It's a lot slower to do the odd-even sort with a small array because the time gain of time that the threads bring does not match the time they take to initialize.



On this figure we can see that the more we add threads the more efficient it is until we reach the 100% usage on each core of the CPU, that slows down our computation time. Also, the more thread you have the more time you lose time on updating the atomic variables (because it creates a queue to access the variable).

Performance analysis

From the different results we obtained earlier we can conclude that you cannot use a parallel algorithm in every case. Indeed, when the array is small there's no point in using a parallel algorithm (either MPI or Pthread). Concerning MPI, the pre-processing (process start, data scattering) and the inter-process communications take up too much time compared to the sequential version that only perform swaps.

Furthermore, when you've got to sort long arrays you have much more inner-swaps than inter-swap which result in huge gain of time compared to the sequential approaches.

Moreover, you can have algorithmic limitations (for example my atomic variable in the pthread algorithm) or hardware limitations that won't allow use any number of processes you want. In my case I had 12 cores, we could see that the best performances I had were between 11 and 13 processes because there was no hardware throttling. If you add up more real cores (for example using another computer) but the gain rate will decrease because of the non-parallelized parts.

Conclusion

In this experiment, the even-odd transposition sort was successfully implemented using MPI and Pthread. The performance analysis shows that both the MPI and Pthread program have a decent performance upgrade on array with a consequent size and using less than 14 processes. The Pthread program have more continuous result but is slower because it lacks optimization.