NOVA SCHOOL OF SCIENCE & TECHNOLOGY

N-Body Problem Using MPI (CAD 2021/2022)

David Neves nº 55539
Rodrigo Mesquita nº 55902

1 INTRODUCTION

The aim of this project is to analyse the performance improvements of parallelizing a *n-body problem* algorithm using MPI, given a sequential implementation as reference. For this purpose, we'll present two parallel algorithms, based on [1], specify various relevant implementation details and optimizations we used, and study their performance using different system configurations.

2 IMPLEMENTATION APPROACHES

The generic approach to parallelize the *n-body problem* starts by partitioning the n particles by p processes, so that each process handles the forces, velocity and position of a subset of $locn = n/p$ particles.

Basic. In the first basic approach, every processes stores the position and mass of every particle, while forces and velocities are only stored and calculated locally. One of the processes is defined to be root (process 0). In the begging of the program the root process initializes all the particles, then broadcasts all the positions and masses, and scatters the initial velocities.

```
MPI_Bcast(mas, N_BODIES, MPI_DOUBLE, ROOT, MCW);
MPI_Bcast(pos, 2*N_BODIES, MPI_DOUBLE, ROOT, MCW);
MPI_Scatter(vel, 2*locn, MPI_DOUBLE, loc_vel,
            2*locn, MPI_DOUBLE, ROOT, MCW);
```

During the simulation, each process processes its *locn* particles using the local velocities and forces arrays, and a `loc_pos` pointer which points to a memory location of the global positions array. Given that every process has access to the position and mass of every particle in the system, when calculating the total force applied on a process's particle, it is possible to do so without any additional communication. At the end of each simulation step an allgather is performed so that every processes has the newly updated positions of all the particles.

```
MPI_Allgather(loc_pos, locn*2, MPI_DOUBLE, pos,
              locn*2, MPI_DOUBLE, MCW);
```

Once the simulation ends, the locally calculated velocities are gathered by the root process (the updated positions are already stored).

Looking at the MPI operations we just presented, it is possible to see that we didn't separate the x and y components of the various vectors being handled in the system in different arrays. We opted to define a structure `vec`, containing two doubles, and storing the positions, velocities and forces in `vec` arrays. Since two doubles have a combined size of 16 bytes (an "efficient" memory size), the `vec` struct isn't subjected to any padding by the compiler. This means that we are able to send and receive the various arrays using the `MPI_DOUBLE` data type without any problem by simply doubling the `count` parameter (two doubles x and y for each element. This choice was made because less MPI communication operations are required if less arrays have to be sent, improving performance since the time it takes to start up a message is substantial [1].

Reduced. The reduced version of the algorithm has some similarities with the basic one, given that we are still partitioning the system's particles in subsets of $locn = n/p$ particles for each process to handle. The main difference here, is that a given process doesn't only calculate the forces applied on its own local particles, it also calculates the corresponding symmetric forces applied on particles that might be handled by another partition/process. Another factor to take in account with this new approach, is the varying iteration length for calculating the total force on each particle. A simple contiguous partitioning as before would yield very imbalanced workloads between the processes, and a stride partitioning works best in this scenario [1]. These new features can result in a quite complex communication protocol between all the processes to achieve the desired results. Pacheco proposes using a phase-communication system during every simulation step so that every process is able to share and receive all the required information/data.

To achieve a strided partition of the particles, when the the positions and velocities are inputed or outputed, a strided index is used instead of the "real" one, allowing us to then simply scatter and gather these arrays to and from the various processes. Figure 1 shows this strided index mechanism considering 3 processes and 9 particles. When the pth particle values are being read, they are stored using the index *stride_p* instead of p, and all the particles that will be processed by the same process, become contiguously stored. The index *stride_p* is calculated as follows (where `size` is the number of processes and `locn` the size of every partition):

```
int stride_p = (p%size)*locn + p/size;
```

`(p%size)` gives us the process that will handle particle `p`. Multiplying this by `locn` gives us the starting index of the partition in the strided array. `p/size` gives us the offset that can be added to the starting index.

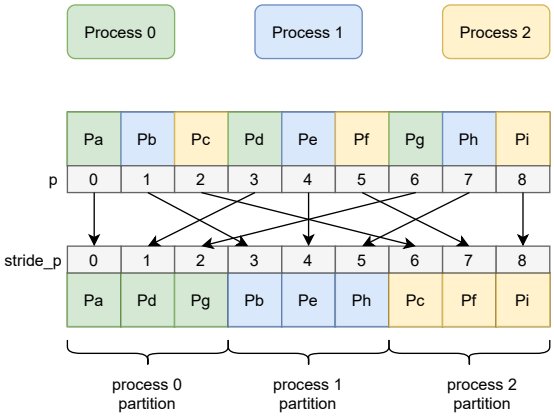

Figure 1: Strided index mechanism.

Once the strided arrays have been created by the root process, we can scatter the positions and velocities and broadcast all the masses. Whenever a process needs to know the mass of a particle , given its owner (rank of the process that owns the particle) and local index, it can do so by using the particle's global index with the following function:

```
int global_index(int p, int owner) {
    return owner + p*size;
}
```

With all the partitioning done, the simulation starts and the algorithm described by Pacheco follows. At the begging of a simulation step, a process copies its local positions to a `tmp_pos` array. It then calculates all the forces between the local particles, storing the symmetric forces in a `tmp_force` array. After this, the phase-communication process happens, where the `tmp_pos` and `tmp_force` arrays are shared between processes and new forces are computed with the receiving particle positions. Finally, the temporary forces are added to the local forces, and the local particles are moved. A relevant implementation detail regarding the phase communication, is the usage of the `MPI_Sendrecv_replace` operation (as suggested by Pacheco) to send/receive the temporary forces and positions arrays in a safe manner. To improve performance, we used the `vec` struct to define the arrays as before, and also joined `tmp_pos` and `tmp_force` into a single contiguous array, resulting in even less MPI communication operations. With these improvements, we only require a single MPI communication call in every phase iteration (and a final send/receive after the loop):

```
MPI_Sendrecv_replace(tmp_pos_force, 4*locn,
    MPI_DOUBLE, dest, rank, source, source,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

To simplify indexing to the `tmp_pos_force` array, we defined two pointers `tmp_pos` and `tmp_force` pointing to the respective memory locations.

In every phase-communication iteration, a process iterates over its local particles calculating the forces resulting from the receiving particle positions. However, only the receiving particles that have a larger non-strided global index must be considered. Given a local particle `q` and a set of positions received from a process whose rank is `owner`, the iteration over the receiving particle positions (to calculate the exterior forces over `q`) is done as follows:

```
int k = (owner <= rank) ? q+1 : q;
for(; k < locn; k++) { ... }
```

If the owner has a smaller or equal rank than the current process, the first particle from the owner that has a larger global index than the global index of particle `q` (index corresponding to the local particles of the current process) must be `q+1` (index corresponding to the local particles of the owner), and `q` otherwise. This can be intuitively verified looking at the locations of the particles "owned" by each process in the original array in figure 1. Lets consider particle *Pe* of process 1, meaning $q = 1$, and process 0 as being the owner of the receiving forces, that is, $owner = 0$. The first particle from process 0 that has a larger global index than *Pe* is *Pg*, which is the third particle owned by process 0, giving us $k = 2 = q + 1$ as we expected since $owner <= rank$.

Regarding memory efficiency, only the root process allocates the arrays for storing all the systems positions and velocities required for IO.

3 EVALUATION

Before presenting the test results, some issues with our reduced solution should be noted. While testing, we found out that the reduced version produced the same results as the basic and sequential versions of the algorithm when using a small number of particles (roughly <= 100), but started showing rounding errors for larger numbers of particles. This is probably due to the reordering of arithmetic operations regarding floating points. The issue could not be mitigated and we concluded that there was no way around these rounding errors given the nature of the algorithm. The algorithm also only works if the number of particles in the system is divisible by the number of processes, giving faulty partitions otherwise. The assignment suggested testing with 17 processes using 2 nodes, but since we used 2048 particles in every test, this results in incorrect values. For this reason we provide benchmarks using both 17 and 16 processes running on 2 nodes, making it clear, that the results obtained using 17 processes were technically incorrect.

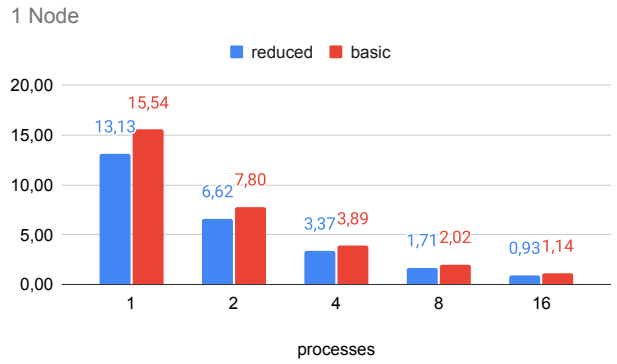

Figure 2: Execution time in seconds running on 1 node.

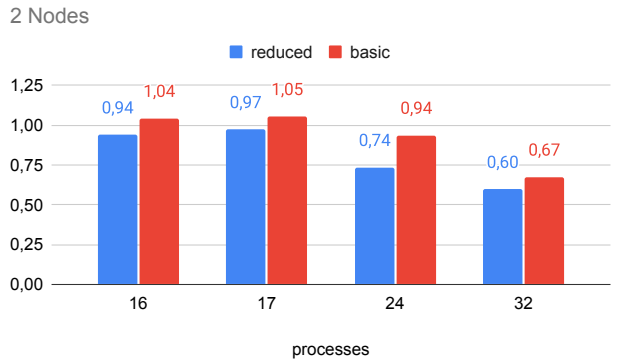

Figure 3: Execution time in seconds running on 2 nodes.

Methodology. All the benchmarks were run on the DI cluster using Bulbasaur nodes in batch mode. Timing the execution times was done using the standard c `clock()` function. The timer starts right before the first MPI communication operation (broadcast) and stops after the last MPI communication operation (gather). We tested the sequential, basic and reduced algorithms using 2048 particles and varying the number of processes and nodes being used in the system.

Result Analysis and Interpretation. The sequential basic algorithm showed a total execution time of 12.613222 seconds while the reduced version only 7.569391 seconds. Looking at figures 2 and 3 we can see that regarding the parallel algorithms, the reduced version also always yields better results. However, the speedup between the reduced and basic versions isn't as significant for the parallel versions as it is for the sequential. This is probably due to the various overheads associated with the extra communication required during the reduced algorithm.

When using just one process, the sequential algorithms perform better than the parallel, which makes sense considering that we are not leveraging any parallelism when using only one process. From 2 to 8 processes we can observe significant speedups (figure 2), while for more than 8, the speedups begin to become less prominent (figure 3).

Our fastest execution time using the parallel reduced algorithm was 0.597614 seconds, meaning a speedup of $S = 7.569391/0.597614 \approx 13$ compared to the sequential reduced implementation, and a speedup of $S = 12.613222/0.597614 \approx 21$ compared to the sequential basic implementation.

4 CONCLUSIONS

We consider that the requirements proposed in the assignment were met. We implemented both the algorithms described by Pacheco following general good coding practices and worrying about implementation details that could affect the final performance. We did notice that the benchmarks achieved by Pacheco, seemed to show a more prominent speedup from the basic to the reduced algorithm using MPI. This could be the case simply because of the different parameters and system configurations being used, or could be related to a lesser efficient implementation from our part. Regardless, the results we achieved still showed significant speedups and were overall in-line with what we expected.

5 RENDER

Out of curiosity, we wrote a simple rendered for the algorithms using the CSFML library. This of course isn't part of the assignment, but since we found the visual results being quite interesting in the context of this project, we thought it would be fun to leave a sequence of renders we made using a system with 4 particles (figure 4).

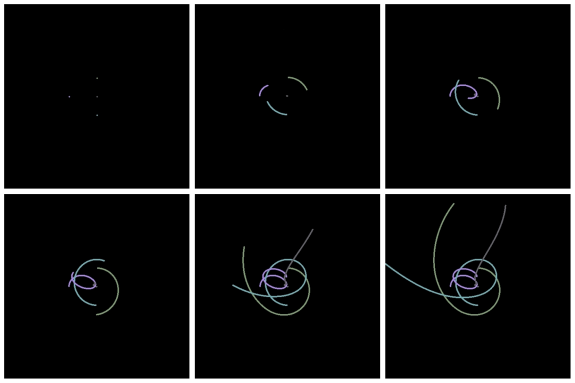

Figure 4: Render of the execution of the n-body algorithm using 4 particles.

REFERENCES

[1] P. Pacheco and M. Malensek. *An Introduction to Parallel Programming.* Elsevier Science, 2021.