MATH-454 - Parallel and High performance Computing - Final Project

MPI and CUDA parallelization of N-body solvers

Francesco Sala, Sciper: 354790 , e-mail: francesco.sala@epfl.ch

EPFL, Lausanne, Switzerland

I. Introduction

In this project, parallel computing is exploited to accelerate the execution of two different solvers for the N-body problem, implemented in C and C++ respectively. In particular, an MPI parallelization of the Barnes-Hut algorithm and a CUDA version of the brute force approach are implemented. For the MPI parallelization, the strong and weak scaling of the code are studied by comparing such scalings with Amdahl's and Gustafson's laws, while an analysis of the execution time against the number of threads per block is carried out for the CUDA implementation.

II. Profiling

As a first step to efficiently parallelize the code, both the Barnes-Hut (BH) and the brute-force (BF) implementations are profiled. As Fig. 1 shows, the bottleneck of the BH algorithm is the `compute_force_in_node` function, which needs to go through the entire tree for each particle to compute the force acting on it. Similarly, in the case of the BF approach (Fig. 2), the bottleneck of the algorithm is the function `compute_brute_force`. For this reason, the MPI implementation of BH is aimed at parallelizing the call to `compute_force_in_node`, while the CUDA version aims at parallelizing `compute_brute_force`.

```
Samples: 2K of event 'cycles:u', Event count (approx.): 2158590221
  Children      Self  Command      Shared Object      Symbol
-   92.39%     0.00%  nbody-code   nbody-code         [.] _start
     _start
     __libc_start_main
   - main
      - 86.13% nbodybarneshut
         + 69.16% compute_force_in_node (inlined)
         + 12.21% move_all_particles (inlined)
         + 2.60% construct_bh_tree (inlined)
         + 2.12% clean_tree (inlined)
        + 6.24% read_test_case
+   92.39%     0.00%  nbody-code   libc-2.28.so       [.] __libc_start_main
+   92.39%     0.00%  nbody-code   nbody-code         [.] main
+   86.13%     0.00%  nbody-code   nbody-code         [.] nbodybarneshut
Tip: Boolean options have negative forms, e.g.: perf report --no-children
```

Fig. 1: Profiling of original Barnes-Hut implementation (40k particles, 4 iterations).

```
Samples: 156K of event 'cycles:u', Event count (approx.): 139644659579
  Children      Self  Command      Shared Object      Symbol
-  100.00%     0.00%  nbody-code   nbody-code         [.] _start
     _start
     __libc_start_main
   - main
      - 99.97% nbodybruteforce
         - 99.96% compute_brute_force (inlined)
              max (inlined)
+  100.00%     0.00%  nbody-code   libc-2.28.so       [.] __libc_start_main
+  100.00%     0.00%  nbody-code   nbody-code         [.] main
+   99.97%    99.95%  nbody-code   nbody-code         [.] nbodybruteforce
+   99.96%     0.00%  nbody-code   nbody-code         [.] compute_brute_force (inlined)
+    4.72%     0.00%  nbody-code   nbody-code         [.] max (inlined)
     0.03%     0.00%  nbody-code   nbody-code         [.] read_test_case
```

Fig. 2: Profiling of original brute-force implementation (40k particles, 4 iterations).

III. MPI implementation

Let N be the total number of particles in the considered "galaxy", and n the number of MPI processes. In the original code, at each iteration, the function `compute_force_in_node` is called recursively starting from the root node. Then, once the recursion reaches an external node, i.e. a node containing a single particle pt is found, the force on pt is computed by calling `compute_force_particle` that introduces the approximation behind the algorithm[1]. In the MPI implementation, each process has a copy of the tree of particles; the call to `compute_force_in_node` is suppressed, and each process updates the forces and positions of only a fraction N/n of particles[2]. In particular, by accessing only a reduced number of entries of the array `array`, each process computes the forces acting only on a portion of particles, and updates their positions accordingly. At this point, the communication takes place: using `MPI_Allgatherv`, 9 variables (doubles: f_x, f_y, f_z, v_x, v_y, v_z, x, y, z) per particle are communicated across all processes so that each process can update its `array` and thus its own tree. Note that it is necessary to use the `v`-version of `MPI_Allgather` since in general, when N is not a multiple of n, the last process will be dealing with a larger number of particles. The necessary buffers for sending and receiving the transmitted message, along with the array of displacements, are suitably defined before the communication and freed at the end of the computation. The MPI parallelization is thus aimed at both the computation of the force and the update of the positions of the particle, that is, as Fig. 1 shows, we have a parallelizable part p that accounts for roughly 81% of the total execution time. Note that this percentage is dependent on the size of the problem, but the value of p is kept constant for the strong scaling to ease the comparison with Admahl's law. Finally, to be able to easily run the code for different values of N, the original `read_test_case` function is changed to read from the specified text file a number of lines equal to the number specified from the command line when executing the code[3].

[1] If a group of particles is far from the given particle pt, such particles and their influence on pt are treated as those of a single mass.

[2] Here the actual division is performed such that, if N is not a multiple of n, the last process has to deal with $N - N \cdot (\mathrm{n} - 1)$ particles.

[3] On the contrary, the number of iterations (4) is not changed either for the MPI or the CUDA implementation.

A. Strong scaling

To assess the strong scaling of the implemented code, a new galaxy made of 500000 particles is generated using a simple Python code (see Appendix A). This is done in order to better study how the program scales. For this scaling, the metric is the speed-up $S(n)$, defined as the ratio of the execution time for one process T_1 divided by the execution time T_n obtained using n processes :

$$S(n) = \frac{T_1}{T_n} = \frac{T_1}{T_1(1-p) + \frac{T_1}{n}p} = \frac{1}{(1-p) + \frac{p}{n}} \tag{1}$$

In the ideal scenario, $p \to 100\%$ and thus $S(n) \to n$; in this project, as already mentioned, we are only parallelizing roughly 80% of the code, and therefore Amdahl's law will predict an upper bound that deviates from the line $S(n) = n$.

B. Weak scaling

In the weak scaling, the workload for each process is kept constant while the number of processes and the dimension of the problem are increased accordingly. The metric for the weak scaling is given by the efficiency $E(n)$. Gustafson introduces the hypothesis that the serial part of a program is the same when changing n, while the parallel part increases when increasing n. Therefore Gustafson's law reads:

$$E(n) = \frac{(1-p(N))T_n + p(N)T_n n}{n((1-p(N)T_n + p(N)T_n)} \tag{2}$$

$$= \frac{(1-p(N)) + p(N)n}{n((1-p(N) + p(N))} \tag{3}$$

$$= \frac{(1-p(N)) + p(N)n}{n} \tag{4}$$

The fraction p is now a function of the dimension of the problem N. In the ideal case, $p(N) \to 100\%$, then $E(n) \to 1$. Now, in this project, the computational cost of BH algorithm scales with $\mathcal{O}(N \log N)$: for this reason, to keep the workload per process constant, a problem size $N = 480000$ is fixed for the maximum number of processes n_{max}. Then, the corresponding workload per process is computed as

$$w = \frac{N \log N}{n_{max}} = \text{const.} \tag{5}$$

Once w is known, it is possible to compute N given n by inverting Eq. 5, i.e. by solving a non-linear equation[4]. The resulting sizes of the problem are $N \in \{10587, 19831, 37283, 70325, 133050, 252450, 480000\}$. The original BH sequential implementation is then profiled for these values of N, so as to plot the observed trend of $E(n)$ along with Gustafson's prediction.

[4]Clearly, the whole procedure can be carried out starting from any value of n, such as a given value of n_{min}.

IV. CUDA implementation

The brute force approach is adapted in C++ to make use of the computational capabilities of a GPU to accelerate the performance of the code. A kernel `compute_force_kernel` is implemented: it takes as input the array of all particles, N and the time step and uses a single thread to update the force over one particle, by looping over the entire array of particles. A 1D grid and 1D blocks are adopted for this scope; the 1D blocks are defined from the command line, and the variable `grid_size.x` is defined accordingly. The command `cudaMallocManaged` is used to allocate an array `d_array` of the same dimension and type as the original array of particles. This array is automatically managed by the Unified Memory system. A subsequent call to `cudaMemcpy` is used to copy the memory of the original array on a new one available to the GPU. At each iteration, a function `compute_forces_and_update_positions` is called to compute the forces using the kernel, synchronize all the threads, and update the positions of all particles. Finally, the array `d_array` is freed.

V. Results

Fig. 3 shows the strong scaling of the BH MPI implementation for different values of N and for a number of processes $n \in \{1, 2, 4, 8, 16, 32, 64\}$, while Fig. 4 shows the weak scaling of the same code, along with the upper bound of Gustafson's law.

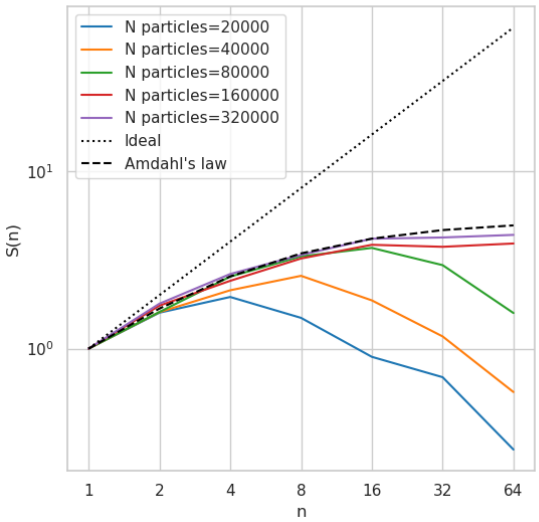

Fig. 3: Strong scaling BH MPI implementation.

Finally, Fig. 5 shows how the execution time per iteration varies against the total number of threads per block.

VI. Discussion and conclusion

A. MPI implementation

The strong scaling results in Fig. 3 show that the parallelization of BH algorithm deviates almost immediately from the

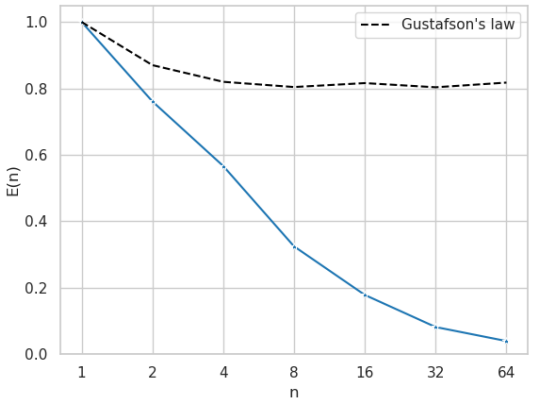

Fig. 4: Weak scaling BH MPI implementation.

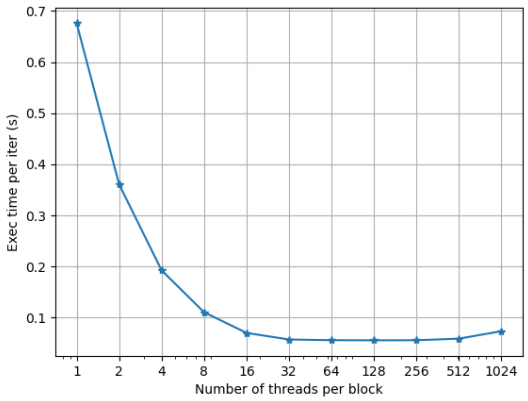

Fig. 5: Execution time per iteration, CUDA implementation of BF algorithm.

ideal trend $S(n) = n$: since we are only parallelizing roughly 80% of the given code, the serial part makes it impossible to reach values of $S(n)$ comparable to the ideal one, that is a code n times faster when using n processes instead of 1. Instead, the code follows the prediction of Amdahl's law, at least for a small number of processes, especially if the problem is larger (larger N). Generally speaking, all the curves show a peak followed by a decrease in speedup, which takes place when the time for communication becomes no longer negligible if compared to the computational time. It is thus possible to conclude that it is not beneficial to enlarge as much as possible the number of exploited processes n, especially for relatively small problems. For example, for a problem of size $N = 20k$, the maximum speedup is reached for $n = 4$; then $S(n)$ drops dramatically. Instead, as N grows, the peak of speedup moves to the right, that is the number of processes that lead to the maximum speedup grows. Moreover, when moving from $n = 32$ to $n = 64$, we are requesting more than one node in Helvetios, thus increasing the communication time. This is not noticeable for $N = 160k$ and $N = 320k$, even though for these values of N, $S(n)$ seems to be reaching a plateau around $n = 64$ processes. A further study should focus on finding the peak speedup for these sizes and verify the probable drop after $n = 64$ or more processes. Besides, for n around 2, the curves for $N = 160k$ and $N = 320k$ are slightly above Amdahl's prediction: it should be kept in mind that we are plotting Amdahl's law for $p = 81\%$, but, as the problem becomes larger, p is likely to grow. Additionally, the presented results are affected by a random error: the experiments could be carried out multiple times to register a more reliable mean trend. Finally, it could be considered to parallelize the I/O function used for reading the initial positions of the particles from a file.

Regarding the weak scaling, Fig. 4 shows a clear divergence between the observed trend and Gustafson's law. While keeping the workload per process constant, the efficiency $E(n)$ quickly deteriorates, passing from almost 0.8 for $n = 2$ to slightly less than 0.1 for $n = 64$. It should be noted that, according to Gustafson's law (Eq. 4), for $n \to \infty$, $E(n) \to p(N) \approx 80\%$. This ideal trend is discernible in Fig. 4. The observed trend allows for the conclusion that, as the problem grows, if you enlarge the number of processes n, the serial part of the code (especially communication) is not fixed. A further study of how `MPI_Allgatherv` scales with n should be carried out to check that communication takes a non-negligible amount of time for a larger number of processes.

The presented results also suggest it could be worth studying a more sophisticated parallelization of the code, so as not to use a blocking communication, to increase p as much as possible, and to improve the weak scaling of the proposed implementation.

B. CUDA implementation

According to the specifications of the Nvidia V100 GPU, the maximum number of Streaming Multiprocessors (SMs) equals 80, while one warp is made of 32 threads. Fig. 5 shows that the execution time per iteration drops by a factor of seven when increasing the number of threads per block up to a size of the block of 32 threads. Then, a constant value is maintained up to 512 threads per block, while a slight increase is observed for 1024 threads. Overall, the total number of threads is always constant ($40k$), and the computation that each thread has to carry out is always the same: the update of the force on a given particle by looping over all the other particles. What changes is how the threads are managed by the GPU. In particular, when the number of threads per block equals 2, a total of 20k blocks is allocated. This means that each SM will have to deal with $20k/80 = 250$ blocks, which is a number larger than the maximum number of resident blocks per SM (32): more than one round of computations has to be performed. Additionally, in this case, the warps are for the largest part idle, since there are only 2 threads per block (thus 2 threads per warp are actually used). Overall, the occupancy of the GPU is low and the execution is not efficient. The situation is similar for blocks of 4 or 8 threads. When the threads per block reach 16, we have $40k/16 = 2500$ blocks, with $2500/80 \approx 32$

blocks per SM. This value is equal to the maximum number of resident blocks per SM: these computations are carried out in "one round" and the execution is efficient, but still we have warps partially idle since there are only 16 threads per block. Conversely, when the number of threads per block reaches 32, then 1250 blocks are created for $1250/80 \approx 16$ blocks per SM, and in this case, the warps are exploited to the full, i.e. 32 threads each. Similarly, when the number of threads per block equals 64, $40k/64 = 625$ blocks are created; each SM deals with $625/80 \approx 8 < 32$ resident blocks, no limitation is encountered due to the specifications of the GPU, and the warps are made of 32 threads. When the blocks are made of 512 threads, a total of 79 blocks are created, i.e. approximately one block per SM is allocated. The performance starts to decrease for a size of 1024 threads per block: a total of ≈ 40 blocks are allocated, that is only half of the available SMs are exploited. To conclude, the implemented CUDA version of the BF algorithm shows that the execution will be faster if the computational capability of the SM is exploited at its fullness: this happens if the warps are fully used, that is 32 active threads are processed in each warp, and if the number of allocated blocks is close to the number of available SMs (80).

APPENDIX A
PYTHON CODE TO GENERATE LARGER GALAXIES

The following Python code is used to generate larger galaxies. Initial positions and velocities are generated according to a Gaussian distribution, in the interval $[-100, 100]$, i.e. values comparable to the ones available in the original `galaxy.txt`.

```
1
2  import random
3
4  # Define the number of lines
5  num_lines = 500000
6
7  # Open the file in write mode
8  with open('large_file.txt', 'w') as f:
9      f.write(str(num_lines) + '\n')
10     for i in range(num_lines):
11         # Generate six random float numbers
12         float_nums = [random.uniform(-100, 100)\
13                       for _ in range(6)]
14         # Write the line to the file
15         f.write('\t'.join(map(str, float_nums\
16                     + [0, i+1, 0])) + '\n')
17
18
19
20
```

APPENDIX B
REPRODUCE THE RESULTS

To reproduce the results, access the folder `project_nbody_MPI` or `project_nbody_GPU`. In both cases, the shell script `tmp.sh` is to be executed from the command line to produce the results in the folder `output/`. This shell script is used to run a Slurm job (either `nbody_MPI.job` or `nbody_GPU.job`) by specifying different number of processes, different dimensions of the considered galaxy (MPI implementation), or different grid size for the CUDA implementation.