DD2356 - Methods in High-Performance Computing

Project:
Quantum Smoothed-Particle Hydrodynamics

Group 11

Valentin de Bassompierre vgomgdb@kth.se 20020727-T198

June 3rd, 2024

1 Introduction

In this project, we implement a numerical solution to the non-linear Schrödinger equation (NLSE) using smoothed-particle hydrodynamics (SPH), a concept presented in Mocz & Succi (2015) [1]. The working concept is that the NLSE can be reformulated under the Madelung transformation to resemble fluid equations. In this form, the NLSE describes the evolution of the quantum probability density of the wavefunction under a quantum "pressure" tensor. SPH is a particle based method well suited to such fluid equations, where interactions between volume elements, such as the pressure gradient, are represented as forces between particles. The two fundamental ideas of SPH are (1) to evolve the positions and velocities of particles according to the calculation of the forces on each particle at each time step, and (2) to use an interpolating/smoothing kernel to calculate forces and spatial derivatives. This method finds applications in a wide range of quantum mechanics problems, including soliton-soliton collision, Bose-Einstein condensates, and collapsing singularities. We focus on the toy example of the simple harmonic oscillator.

More precisely, we implement a C version of the following Python code https://github.com/pmocz/QuantumSPH [2], and characterize the final code's computation and memory system performance. Further, we develop a shared-memory version of the code using OpenMP as well as a distributed version of the code with MPI. The aim of the project is to use performance monitoring and profiling tools to identify bottlenecks, and optimize the code's performance accordingly. Our code, including OpenMP, MPI, and comparative versions, can be found at https://github.com/Valentin-dB/DD2356_QuantumSPH.

2 Experimental Setup

Throughout the entirety of this project we analyse the performance of our code on the Dardel supercomputer [3]. It is an HPE Cray EX supercomputer installed at PDC Center for High Performance Computing at the KTH Royal Institute of Technology. We use the CPU partition, where each compute node is equipped with two AMD EPYC™ Zen2 2.25 GHz 64-core processors. Two virtual hardware threads are enabled on each physical CPU core, which means that each compute node has a total of 256 logical/virtual cores.

To monitor the performance of our code and carry out it's profiling, we use the perf tool, a performance counter which we use to count hardware events such as instructions executed or cache-misses suffered. By computing ratios, this can help us access more relevant information, for instance the number of instructions per cycle or the cache miss ratio. We also use CrayPat for profiling, which measure metrics such as the time spent on each function or the imbalance between our different threads or processes.

3 The Serial Code

3.1 Initial performance analysis and profiling

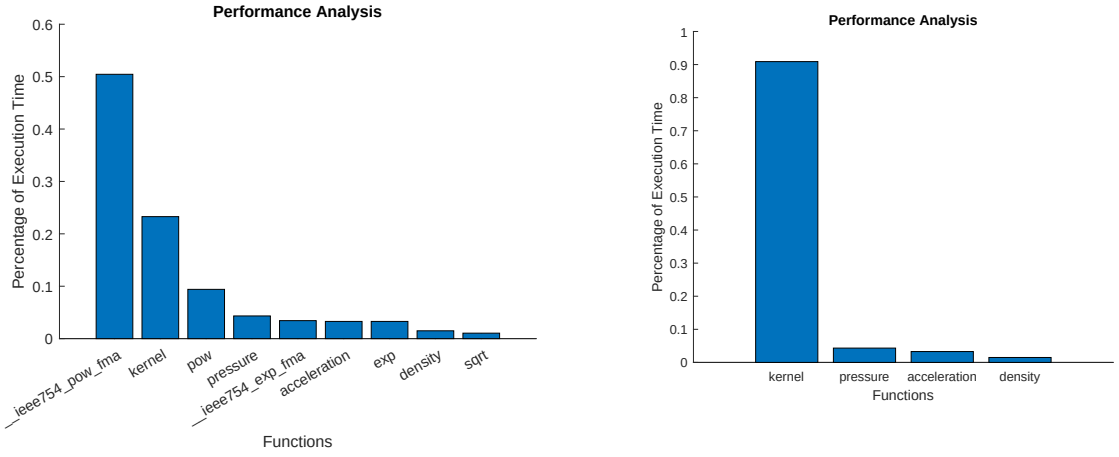
We first develop a serial version of the code. Before implementing any optimizations, we monitor the performance of our code to see which functions take up the most time, and evaluate whether they are compute bound, memory bound, or I/O bound. To start of, we use the perf tool.

The first thing we notice is that the L1 cache miss ratio is very low, inferior to 0.01%. This indicates that we have a good usage of our memory space, we are re-using data that is already in the cache a lot. This makes sense since our code requires quite little memory in total. All

our data is stored in less than a dozen vectors, each of which contain n elements - the same as the number of particles. With n in the order of the hundreds, they should all fit in the L1 cache (for a 512 kB L1 cache, we can go up to $n \approx 9000$ with all the vectors still fitting in the cache).

Second, we notice that the CPU is almost fully utilized (99.8% utilization), with a high number of instructions per cycle, 2.43. These two observations seem to indicate that the code is compute bound rather than memory or I/O bound.

We further profile our code using CrayPat. We use it in sampling mode, and in total 673 samples were drawn. Figure 1a shows the raw results from CrayPat. As one can see, this includes functions from the math header file (`pow`, `exp`, `sqrt`), as well as some associated internal functions. Looking at our code more closely, we see that we only call these functions inside our `kernel` function (which computes pairwise interactions between particles). Thus, we can clean up the profiling results by considering only our high level functions, associating all the samples of math header file functions with the `kernel` function. Figure 1b shows this cleaned up profiling.



(a) Raw results

(b) Cleaned results

Figure 1: Profiling of the serial code with CrayPat

We see that the `kernel` function takes up more than 90% of our execution time, so it's a major bottleneck. This makes sense as all our higher level functions call `kernel` many times (at least n^2 times) during their execution. Therefore, we should focus on optimizing the performance of this function and/or parallelizing our calls to it. Note also that these results confirm the code is compute bound, as the `kernel` function is only doing compute operations. Our I/O functions take so little time compared to the rest of the execution that they do not even come up in the CrayPat profiling.

3.2 Optimizing our serial code

Based on our results from profiling, we concentrate on optimizing the `kernel` function. We notice (Figure 1a) that the `pow` function is taking a lot of time, even though it is called as many times as the `exp` function or the `sqrt` function. This seemed strange to us as raising a number to the power of a small integer, as is our case (we are raising to the power 5 at most), should be relatively inexpensive to compute. It turns out that this function is constructed to support raising numbers to powers that are not integers, and thus it is quite inefficient for our usage. With that in mind, we replace this function call with manual implementation of the power operation. Further, we reduce computation cost by storing intermediary results (such as h^2 and r^2) and constant values we use repeatedly (such as $1/\sqrt{\pi}$). This effectively reduces the number of

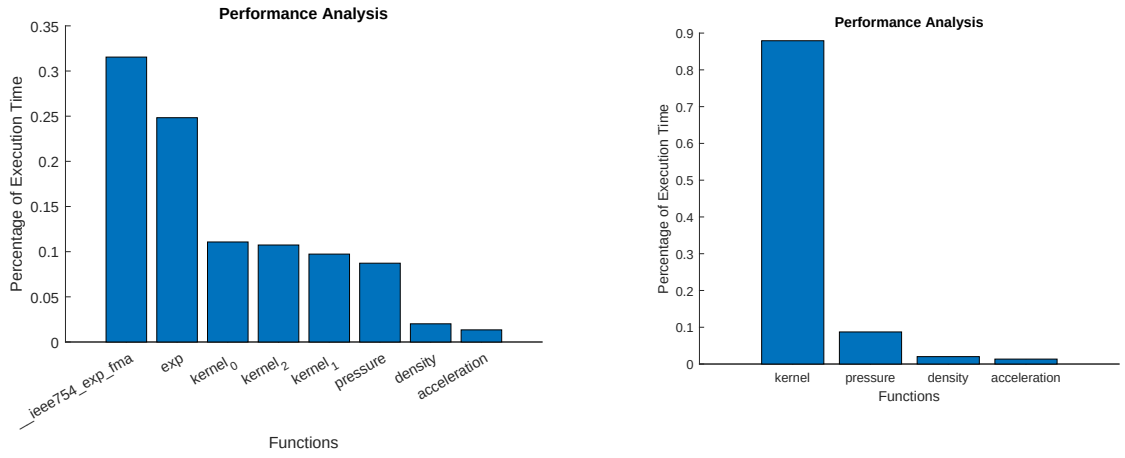
operations we have to execute. Finally, we notice that our `kernel` function is used to compute the basic smoothing kernel, but also it's derivatives up to the second order. In our initial code, this was handled by using the `switch case` statement and an input argument indication which derivative order to compute, meaning there was conditional statement overhead each time we called the `kernel` function. To remove this overhead, we implement instead 3 kernel functions, one for each order of derivative.

For 200 particles and 1 000 time steps (800 for the set up and 200 for the actual simulation), these improvements reduce the average execution time (over 10 runs) from 6.704 seconds (with 0.003 seconds standard deviation) to 2.387 seconds (with 0.002 seconds std). Looking at the perf tool (see Table 1), we see that we have managed to reduce the total number of instructions by roughly 65%, at very small to no memory cost. The L1 cache miss ratio is still less than 0.01%. Further, the number of instructions per cycle remains almost unchanged. Thus, although our small improvements have yielded substantial gains, the code remains compute bound.

Version	Initial	Improved
Instructions [billions]	55.29 (100%)	19.40 (35.1%)
Instructions per cycle	2.43	2.40
CPU utilization	99.8%	99.4%
L1 cache miss ratio	0.01%	$< 0.01\%$
Elapsed time [s]	6.704 (100%)	2.387 (35.6%)

Table 1: Perf results for our initial and improved serial code

We use CrayPat again to profile our improved version of the code. The results are shown on Figure 2:



(a) Raw results

(b) Cleaned results

Figure 2: Profiling of the optimized serial code with CrayPat

We see that the `kernel` function is now taking a bit lower percentage of the total execution time (87.9%), but is still the main bottleneck. However, we notice that during the execution of the program, we are computing the interactions between the same particles multiple times. Thus, we might be able to gain some time if we store the results in a matrix (of size $n \times n$), and fetch them at subsequent calls. Further, since entry i, j of the matrix is the interaction between particle i and particle j, it will be symmetric (or anti-symmetric in the velocity case), so we only need to compute the top half of the matrix. Table 2 show the relevant results from perf for all three versions of the code.

Version	Initial	Improved	Memorizing
Instructions [billions]	55.29 (100%)	19.40 (35.1%)	9.70 (17.5%)
Instructions per cycle	2.43	2.40	2.06
CPU utilization	99.8%	99.4%	99.1%
L1 cache miss ratio	0.01%	$< 0.01\%$	0.75%
Elapsed time [s]	6.704 (100%)	2.387 (35.6%)	1.408 (21.0%)

Table 2: Perf results for all three versions of the serial code

We see that this optimization has further reduced the total number of instructions, and the execution time. However, we notice that the L1 cache miss ratio has increased a little. Although this is still a very reasonable cache miss ratio, this increase and the fact that the number of instructions per cycle has dropped a little shows that we are beginning to suffer from memory latency.

Profiling with CrayPat again, we see that calling the `kernel` function remains the main bottleneck, although the percentage of time calling it has now dropped to 67%. As no more serial optimizations are evident, the next step is to parallelize our code so that we can execute multiple calls to `kernel` at the same time.

4 Shared memory parallelism with OpenMP

We proceed to parallelize our "memory" version of the code with OpenMP, in a shared memory environment. Under the "Original OMP" column of Table 3, you can find the perf results for our code with 16 threads, 800 particles and 1 000 time steps. Unfortunately, we see that the number of instructions per cycle is extremely low, only 0.24. We see that we suffer from severe memory latency with more than 80% of cycles stalled in the back-end. We also have a high cache miss ratio (over all levels of cache, including L1 cache), and a high number of cache references overall (almost twice as much as for our serial version with the same settings). All of this indicates that our memory usage is particularly poor. Indeed, in this version of the code the threads access the memory in a dynamic fashion, which means the same cache line is often shared among multiple threads. This leads to more cache references as the lower level caches are not necessarily shared among the threads, so the same data has to be loaded to multiple L1 caches. It also leads to more cache misses as each thread does not access the data in a consecutive manner, so there is little to no spatial locality. Further taking into account false sharing problems (with threads having to update the data multiple times to make sure it wasn't overwritten by another thread writing to the same cache line) leads to the observed performance.

Version	Original OMP	Cache-Blocked OMP
Instructions [billions]	88.14	91.05
Instructions per cycle	0.24	0.56
L1 cache miss ratio	6.23%	2.39%
Cache miss ratio	31.53%	23.36%
Cache references [millions]	3871	1747
Back-end cycles idle	83.67%	65.63%
Elapsed time [s]	6.757	3.017

Table 3: Perf results for our "memory" OpenMP codes

To improve the memory usage we thus implement a cache-blocked version of our code. When computing our matrices we make sure to compute them subblock by subblock, with the

threads each assigned to subblocks instead of individual elements. The results from the perf tool for this version of the code with subblocks of size 32×32 are also shown on Table 3. As one can see, this reduces all the negative effects described in the previous paragraph, thanks to increased spatial locality and reduced cache line sharing between threads. However, the performance is still far from ideal, with only 0.56 instructions per cycle and 65% of back-end cycles stalled.

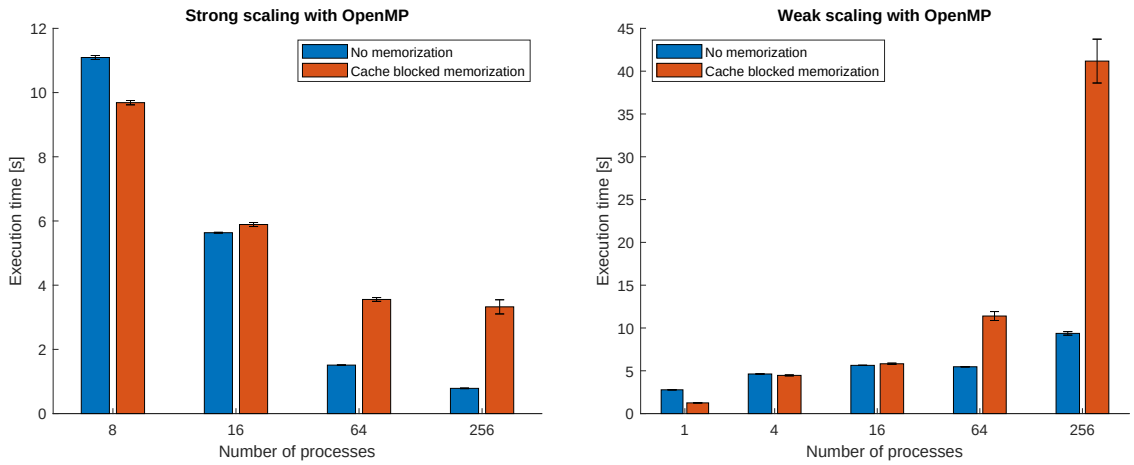
Since this code has so much memory latency (it is quite clearly memory bound), we decide to see whether not memorizing the particle's interactions but simply computing them again might perform better. The results for this code are shown on Table 4.

Version	Original OMP	Cache-Blocked OMP	No memory OMP
Instructions [billions]	88.14	91.05	257.08
Instructions per cycle	0.24	0.56	1.56
L1 cache miss ratio	6.23%	2.39%	0.01%
Cache miss ratio	31.53%	23.36%	23.50%
Cache references [millions]	3871	1747	32
Back-end cycles idle	83.67%	65.63%	6.06%
Elapsed time [s]	6.757	3.017	3.083

Table 4: Perf results for all three versions of the parallelized code

Observe that this change leads to a big increase in the total number of instructions. However, the huge memory latency decrease compensates for this fact, so much so that the cache-blocked version and this new version have roughly the same execution time for these settings. (Note that the overall cache miss ratio is almost constant but the number of overall cache references is far smaller, so the same ratio indicates a far smaller amount of cache misses)

Both versions perform differently for other settings however. Figure 3 shows scaling results for the cache-blocked version of our code as well as the "no memory" version of the code. For the strong scaling, we use 800 particles and 2 000 time steps. For the weak scaling, we use 200 particles and 2 000 time steps for one thread, then double the number of particles every time we multiply the number of threads by 4 (as the complexity is n^2).



(a) Strong scaling

(b) Weak scaling

Figure 3: Scaling results for our OpenMP codes

We observe that for very small number of threads (< 16), the cache blocked version performs better than the "no memory" version (note the typo in the graphs x-axis label, which should

read "Number of threads"). However, it doesn't scale well, likely because an increasing number of threads means increased memory problems (as well as thread maintenance overhead). Also, even for very small number of threads, the difference with the "no memory version" is not as striking as it was for the serial code. In fact, we see on Table 5 that the speed-up is not very close to the number of threads. This is not true for the "no memory" version which performs well under strong scaling, as shown on Table 6 (except for 256 threads, but this is probably because the number of particles, 800, is too small for that many threads to be utilized at their full potential).

Number of threads	8	16	64	256
No memorization	3.75	7.39	27.56	52.96
Cache Blocked	4.30	7.07	11.71	12.52

Table 5: Strong Scaling : Speed-up with respect to the "memory" serial code

Number of threads	8	16	64	256
No memorization	8.21	16.16	60.29	115.85
Cache Blocked	9.40	15.46	25.61	27.38

Table 6: Strong Scaling : Speed-up with respect to the "no memory" serial code

Number of threads	4	16	64	256
No memorization	27.07%	22.23%	22.90%	13.37%
Cache Blocked	27.43%	19.23%	9.68%	2.08%

Table 7: Weak Scaling : Efficiency with respect to the "memory" serial code

Number of threads	4	16	64	256
No memorization	60.10%	49.35%	50.84%	29.68%
Cache Blocked	60.90%	42.68%	21.49%	4.61%

Table 8: Weak Scaling : Efficiency with respect to the "no memory" serial code

Both versions of the code perform less well for weak scaling, although the "no memory" version is still much better for a large amount of threads. One reason might be that all our vectors have to effectively be stored in a higher level of cache shared by all threads, when they could be stored on the L1 cache when only one thread was updating them.

5 Distributed memory parallelism with MPI

5.1 Implementation and performance analysis

We now proceed to parallelize our code with MPI, enabling distributed memory parallelism and thus, running our program on multiple nodes of the Dardel computer. Hopefully, our "memory" version of the code might be put to better usage. Indeed, as each process has it's own local memory, we can hope for less memory sharing problems as encountered in section 4, and thus less memory latency. The working concept of our MPI code is that each process is assigned it's own set of particles, and will compute the density, pressure, velocity, and acceleration associated with these particles. In order to compute interactions between particles, each process will, one at a time, broadcast it's particles locations (or relevant information), so that every process can then compute the interactions between these broadcasted particles and it's own particles. On Table 9, you can see the perf results for this code with 16 processes (on one node), 800 particles and 1 000 time steps, compared to those for the OpenMP versions of our code.

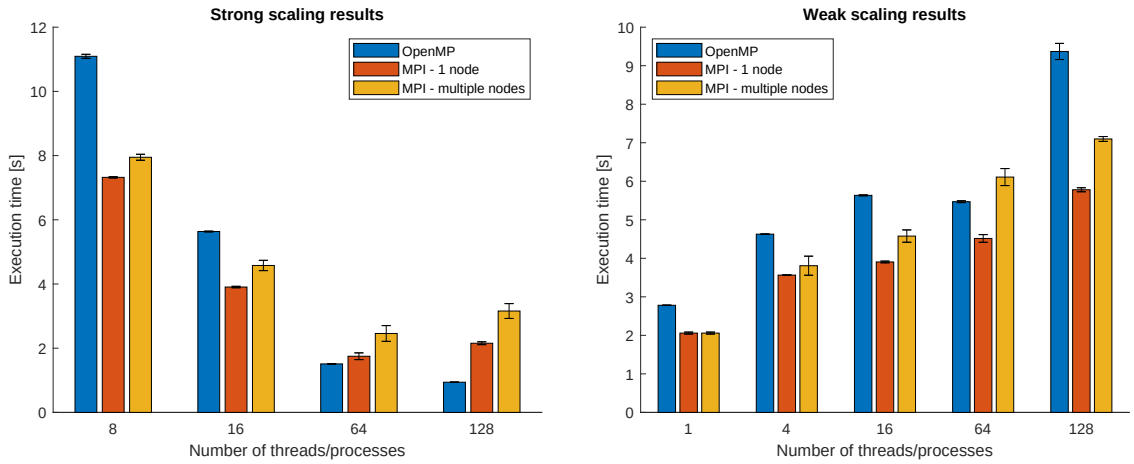
As predicted, we have much better memory usage than with the OpenMP cache-blocked version of the code, enabling us to reach 1.80 instructions per cycle. We see that we have less cache references, less cache misses and thus far less cycles stalled in the back-end. This is because each process's memory space is independent and separated from the other's, so there are no problems from memory sharing. Note however that with the MPI version of the code, we are not taking advantage of the fact that our matrices are (anti-)symmetric, and we are computing interactions between the same two particles twice (once when the first particle is broadcasted, and a second time when the second particle is broadcasted). This leads to a higher number of instructions in total. It might also be a reason why the memory usage is better, as we don't

Version	MPI	Cache-Blocked OMP	No memory OMP
Instructions [billions]	190.68	91.05	257.08
Instructions per cycle	1.80	0.56	1.56
L1 cache miss ratio	0.70%	2.39%	0.01%
Cache miss ratio	14.05%	23.36%	23.50%
Cache references [millions]	869	1747	32
Back-end cycles idle	16.11%	65.63%	6.06%
Elapsed time [s]	1.977	3.017	3.083

Table 9: Comparison of the perf results for MPI parallelism and OMP parallelism

need to access data in very different places of the matrix at the same time (on a side note, it would be interesting to explore an OpenMP version of the code which also does not take into account the symmetry in our matrices, to see if we have better memory usage). We also observe that with this reduced memory latency, we are able to out-perform the "no memory" OpenMP version of our code, at least for the selected settings.

We now perform scaling tests on one node of the Dardel supercomputer, as well as on multiple nodes. As for the OpenMP code, we use 800 particles and 2 000 time steps for the strong scaling, and for the weak scaling, we use 200 particles and 2 000 time steps for one process, then double the number of particles every time we multiply the number of processes by 4. We compare the results with our OpenMP "no memory" version of the code, and plot them on Figure 4. We also indicate speed-up and efficiency on Table 10 and Table 11. For MPI with multiple nodes, we use 4 nodes when using 4, 8, and 16 processes, and 8 nodes when using 64 or 128 processes (with evenly distributed amount of processes per node). Due to the difficulty of obtaining sole usage of multiple nodes, we share the nodes with other users.



(a) Strong scaling

(b) Weak scaling

Figure 4: Scaling results for our MPI code

Number of processes	8	16	64	128
One node	7.99	14.98	33.44	27.15
Multiple nodes	7.36	12.78	23.79	18.52

Table 10: Strong Scaling : Speed-up with respect to MPI with one process

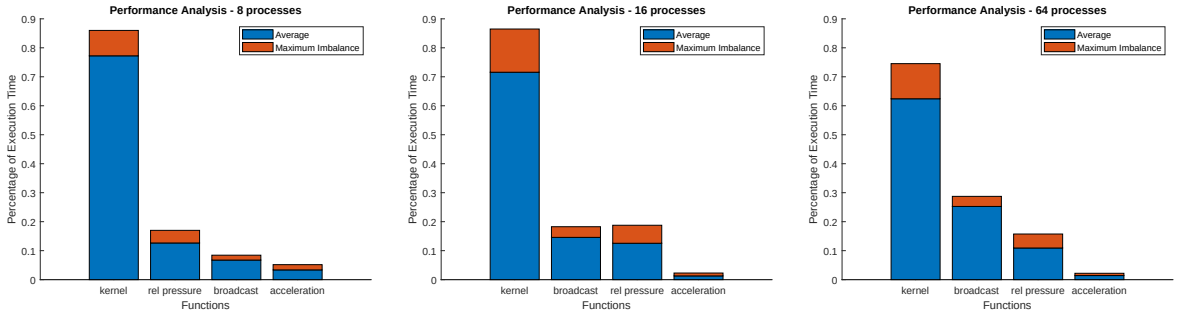
Number of processes	4	16	64	128
One node	57.76%	52.72%	45.59%	35.62%
Multiple nodes	54.07%	44.98%	33.70%	29.00%

Table 11: Weak Scaling : Efficiency with respect to MPI with one process

As one can see, our MPI code generally outperforms our OpenMP implementation. We also observe that our MPI code produces better and more consistent results on one node than on multiple. This might be because the communication is better on one node (as the memory is physically closer), but this might also be due to the fact that when using multiple nodes, we are sharing them with other users which creates interferences.

When there is too few particles per process however (as is the case for the strong scaling case with 64 processes and 12 to 13 particles per process or with 128 processes and 6 to 7 particles per process), the communication and synchronisation overhead becomes more important, and the OpenMP version performs better. In fact, we see on Table 10 that the MPI code has a good speed-up for 8 and 16 processes, but not so good for 64 or more processes (further, for this setting 128 processes is worse than 64). Utilizing large amounts of processes is more useful when we have more particles in total, as we can see with the weak scaling. However, using more processes still creates additional overhead, and is not 100% efficient. Indeed, if p is the number of processes, for the same number of computations (n^2/p), we have more broadcasting operations (p), and less computations per broadcasting operation ($(n/p)^2$). Also, we need to broadcast to and synchronize with a larger amount of process, as well as simply maintain more processes.

Figure 5 shows this additional overhead as the number of processes increase. It plots the profiling results of our code with different numbers of processes (the blue bars are the average percentage of time spent by each process on the specified function, while the orange is the maximum time one process spent on the specified function, scaled in the same way the average time was scaled). Note that for the MPI version, we merge the `density` and `pressure` functions in order to avoid one round of broadcasting operations. The new function is called `relative_pressure`. We see clearly that the broadcasting operation is taking a larger and larger percentage of the execution time as the number of processes increases.



(a) 8 processes (b) 16 processes (c) 64 processes

Figure 5: Profiling of our MPI code, for different amounts of processes

5.2 Development of a performance model

Let us now develop a performance model for this MPI version of the code, and see how it compares to the profiling results. We know that for each time iteration, we need to compute the relative pressure for all particles, and their acceleration. Then we also update their speed and position with a leap-frog method. Let us denote by n the number of particles, n_t the number of time steps, p the number of processes, and $m = n/p$ the number of particles assigned to each process. Let us fist estimate the cost of one broadcast operation. We consider that the broadcast operation is executed in the following way: the broadcasting process sends the message to one other process, then both processes send the information to other processes, the number of processes having received the information doubling at each iteration (in a balanced binomial tree scheme, see [4]). Then, the entire broadcasting would require $log_2(p)$ consecutive sends. We

further consider that there is an initial latency t_{lat} (for process synchronization and preparation) before the first send, and then model the time for one send as m/b, where b is the bandwidth (and $m = n/p$ is our message size). From previous assignments, we had $t_{lat} = 7.83\,\text{e-}6\,\text{s}$ and $b = 2.963\,e9\,s^{-1}$ (equivalent to 23 707 MB/s). Thus, the broadcasting operation would cost:

$$Bcast(m) = t_{lat} + log_2(p) * m/b$$

Next, we model calls to our kernel functions. Depending on whether we are calling the basic kernel, it's first or second derivative, we have a different amount of operations. However, they all call the math `exp` function. Since it's unknown to us the exact algorithm used for this function, we measure the execution time of calling it 1 million times, and use this as a reference. From our tests, we get $t_{exp} = 4.933\,\text{e-}9$ seconds (with the time of one multiply or add operation measured at $t_{flop} = 7.46\,\text{e-}10$ seconds). One call to the kernel function can thus be modelled as:

$$ker_0 = t_{exp} + 4\,t_{flop}$$
$$ker_1 = t_{exp} + 7\,t_{flop}$$
$$ker_2 = t_{exp} + 8\,t_{flop}$$

The computation of the relative pressure starts off with computing the density, as well as it's first and second derivatives. This requires each process to broadcast it's current particles locations, and all processes to compute pairwise interactions (for all three kernels) between the broadcasted particles and their own particles. Then, using these results we compute a factor ($8\,t_{flop}$) for each particle and add-up (pre-computed) pairwise interactions to the relative pressure. Amounting for all other operations inside our for-loops, and neglecting data movement, the computation of the relative pressure can thus be modelled as:

$$RP_c = p * \big(Bcast(m) + m^2 * (ker_0 + ker_1 + ker_2 + 4\,t_{flop})\big) + m * n * 2\,t_{flop} + m * 8\,t_{flop}$$
$$RP_c = p * \big(t_{lat} + log_2(p) * m/b + m^2 * (3\,t_{exp} + 23\,t_{flop})\big) + m * n * 2\,t_{flop} + m * 8\,t_{flop}$$

If we account for data movement as well, we note that we first have to store data in four vectors (initializing them to 0). Then, for each broadcast operation, we have to store results in two kernel matrices (basic kernel and first derivative kernel). Finally, we have to read from the basic kernel matrix (we suppose data is not in the cache anymore as the matrix is quite big). We neglect other data movement because they concern vectors being re-used, for which we suppose the data is in the L1-cache and data movement is fast. From the STREAM benchmark, we get that the read/store bandwidth is 28 836.5 MB/s, such that the time for one load operation is $t_{store} \approx t_{load} = 2.7743\,\text{e-}10\,\text{s}$. Thus, our model becomes:

$$RP_d = m * 4\,t_{store} + p * m^2 * 2\,t_{store} + m * n * t_{load}$$
$$RP = RP_c + RP_d$$

In a similar fashion, the computation of the acceleration can be modelled as:

$$AC = p * \big(Bcast(m) + m^2 * 4\,t_{flop}\big) + m * 3\,t_{flop} + m * t_{store} + p * m^2 * t_{load}$$

Finally, the leap-frog updates of our particles speed and position takes up 6 multiply/adds, thus:

$$LF = m * 6\,t_{flop}$$

The total execution time can then be modelled as:

$$T = n_t * (RP + AC + LF)$$
$$T = n_t * \Big(p * \big(2\,t_{lat} + 2\,log_2(p) * m/b + m^2 * (3\,t_{exp} + 27\,t_{flop})\big) + m * n * 2\,t_{flop} + m * 17\,t_{flop}$$
$$+ m * 5\,t_{store} + p * m^2 * 2\,t_{store} + m * n * t_{load} + p * m^2 * t_{load}\Big)$$
$$T = n_t * \left(2\,p * t_{lat} + 2\,log_2(p)\frac{n}{b} + \frac{n^2}{p}(3\,t_{exp} + 29\,t_{flop}) + 17\,\frac{n}{p}t_{flop} + 5\,\frac{n}{p}t_{store} + 4\,\frac{n^2}{p}t_{store}\right)$$

with the first two terms being the communication cost (increasing with the number of processes), and the last four being the computational cost (decreasing with the number of processes).

On Table 12, you can see our model's predictions compared to the actual profiling from CrayPat (800 particles, 2000 time steps for 8 and 16 processes, 4000 time steps for 64 processes).

Number of processes	8	16	64
Model predictions - Computation part [s]	6.010	3.005	1.502
Measured time - Computation part [s]	7.004	3.492	3.433
Model predictions - Communication part [s]	0.254	0.505	4.022
Measured time - Communication part [s]	0.507	0.596	1.160

Table 12: Model predictions compared to measured execution times

Unfortunately, despite our best efforts, we see that our model's predictions are not very accurate. While the computational part is qualitatively correct for 8 and 16 processes, it is completely off for 64 processes. This is probably because we need to account for more memory latency. Indeed, the compute terms are much more significant than the memory terms (perhaps our estimation of $t_{store} \approx t_{load}$ is not very accurate).

For the communication, our model is also not very useful. The most significant part comes from the initial latency, with the message sending part almost irrelevant in our model. However, this is probably not the case in reality. Especially for fewer processes, where the number of particles per process is higher, the message sending part should also be taken into account. Further, the latency we set (which was based on benchmarking from point-to-point communication) seems to be too high. Maybe another broadcasting model (linear broadcasting instead of tree-broadcasting) would fit our application better. We tried a few other approaches, none yielding convincing results.

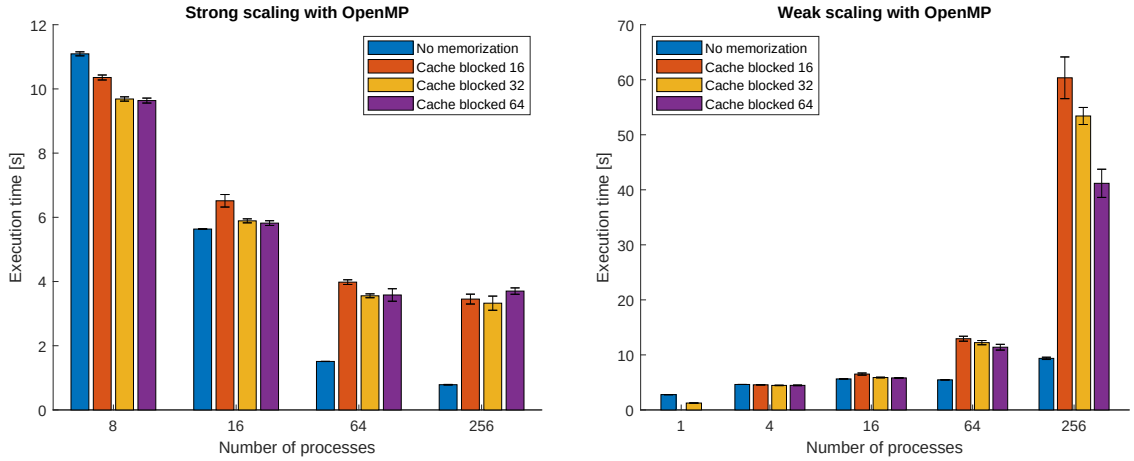
6 Conclusion

In conclusion, we have managed to optimize our compute bound serial code up to the point that memory latency effects could be seen. We've also managed to parallelize our code in a shared memory environment with OpenMP, and in a distributed memory environment with MPI. With the use of performance monitoring tools, we've demonstrated that, at least for relatively small numbers of threads/processes, our parallel implementations are efficient. However, there are still avenues for improvement. First of all, our performance model is inaccurate and if we had more time, we should look further into why that is (perhaps useful information can be found in [4]). One major improvement left to implement is a hybrid version of the code using MPI and OpenMP. This seems like a promising improvement as the MPI verion of our code performed better than the OpenMP, but scaled badly for a large number of processes. A hybrid version of the code would permit better memory usage from MPI while also enabling larger scale parallelism.

Annex

Exploring different cache-blocking sizes

Here we show scaling results for cache blocked memorization in OpenMP, for different subblock sizes (16×16, 32×32, and 64×64 doubles respectively).



(a) Strong scaling

(b) Weak scaling

Figure 6: Scaling results for different cache-blocking sizes

We see that in general 16 is too small a size, while 64 performs best when the number of particles is big. Here it doesn't really matter as the OpenMP code without memorization performs better, but this shows that choosing the cache blocking size is important and not trivial.

References

[1] Philip Mocz and Sauro Succi. "Numerical solution of the nonlinear Schrödinger equation using smoothed-particle hydrodynamics". In: *Physical Review E, Volume 91, Issue 5, id.053304* (2015). URL: https://ui.adsabs.harvard.edu/abs/2015PhRvE..91e3304M/abstract.

[2] Philip Mocz. *Schrodinger Equation Smoothed Particle Hydrodynamics (SPH) Method Solver*. 2017. URL: https://github.com/pmocz/QuantumSPH.

[3] *PDC Center for High Performance Computing - About the Dardel HPC system*. URL: https://www.pdc.kth.se/hpc-services/computing-systems/about-the-dardel-hpc-system-1.1053338.

[4] Emin Nuriyev, Juan-Antonio Rico-Gallego, and Alexey Lastovetsky. "Model-based selection of optimal MPI broadcast algorithms for multi-core clusters". In: *Journal of Parallel and Distributed Computing* 165 (2022), pp. 1–16. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2022.03.012. URL: https://www.sciencedirect.com/science/article/pii/S0743731522000697.

[5] *Perf: Linux profiling with performance counters*. URL: https://perf.wiki.kernel.org/index.php/Main_Page.

[6] *LUMI documentation : Cray Performance Analysis Tool*. URL: https://docs.lumi-supercomputer.eu/development/profiling/perftools/.

[7] *OpenAI - ChatGPT*. 2024. URL: https://www.openai.com/.