Many Body Simulation with MPI

Parallel Computation (CSE 260), Assignment 3

Andrew Conegliano (A53053325) Matthias Springer (A99500782) GID G-338-665

March 14, 2014

Assumptions

- The number of bins in x/y dimension is the same. The original program does not provide a way to specify separate values, though our implementation could cope with that situation.
- Particles do not jump across multiple bins.

Notation

In this work, we use the following notation and variable names.

- M: number of bins in x/y direction, resulting in M^2 bins.
- p_x: number of threads in x direction.
- p_y: number of threads in y direction.
- $np = p_x \times p_y$: total number of threads.
- n: number of particles

1 Runtime Environment

We optimized our implementation for a specific runtime environment. All benchmarks and performance results are based on the following hardware and software.

- Intel Xeon E5354 @ 2.33GHz (*Clovertown* processor)
  - 2 *Woodcrest* Core2 dies
  - 2 sockets per chip
  - Supports SSE, SSE2, SSSE3

1.1 Software

- CentOS 6.3 Linux (Kernel `2.6.32-358.18.1.el6.x86_64`)
- Built with GCC 4.7.3

2 Basic Algorithm

Our project simulates the movement of particles in a 2D space. Particles influence other particles via gravitational force. There is a maximum gravitational force (cut-off), after which the force does not increase anymore. Particles are structured in a 2D bin table, where only particles from the same and from neighboring bins affect particles in the current bin. This is based on the idea that the gravitational force decreases quadratic with the distance. When particles hit the border of the world, they are being reflected.

3 Optimizations

In this section, we present optimizations, that increased the performance of the basic algorithm.

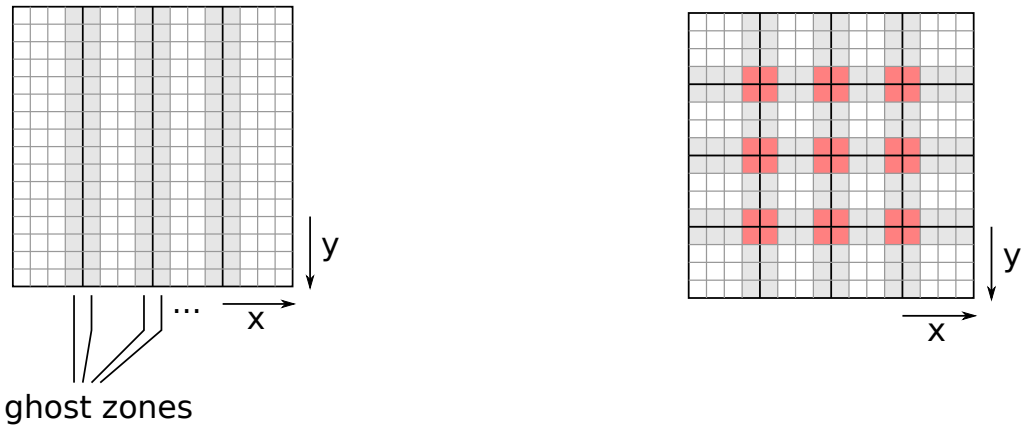
3.1 1D Partitioning

Our first implementation was to distribute the calculation of particles onto multiple threads by using MPI. We divided the world into *np* equally sized stripes (vertically/column-wise), where *np* is the number of threads. Every thread is responsible for calculating the movement of the particles within its stripe.

Movement of Particles The world is already clustered in 2D table of bins. Therefore, we actually delegate the calculation bins to separate threads instead of single particles. A particle can never jump across multiple bins, i.e. it either stays in its current bin or moves to a neighboring bin. Therefore, for every bin that is located at the border of a stripe (i.e. the outermost bins), we have to consider sending particles to a neighboring thread in case a particle crosses over to a bin that is not located on the current thread.

Ghost Zone The gravitational force on particles is limited to particles within the same bin and particles from neighboring bins. Therefore, every thread has to keep track of all particles in immediately neighboring bins of the neighboring threads. Therefore, we have to transfer two columns of bins to every thread, except for the outermost threads.

Message Passing Note, that the non-local bins, i.e. the bins that are not located on the current thread, that we have to consider for movement of particles are the same as for the ghost zones.



(a) Ghost zones with 1D partitioning.

(b) Ghost zones with 2D partitioning.

Figure 1: Ghost zones with 1D and 2d partitioning. Gray ghost zones are sent to 1 neighbor, red ghost zones are sent to 3 neighbors.

Therefore, when calculating the amount of communication, we can focus on the number of neighbors. For the 2D case we will have some additional overhead, because some ghost zones must be sent to multiple threads. The following expression is the number of bins that have to be sent in total.

$$2 \times (p_x - 2) \times M + 2 \times M = 2 \times M \times (p_x - 1)$$

Note, there are two threads that have only one column as a ghost zone. The other threads have two column as a ghost zone. Every column consists of M bins. The number of bins per thread that must be sent is in the order of $2M$.

Implementation Details In our first implementation, we sent every particle separately to the destination thread. At the end of the sending phase, the thread sends a dummy particle. In the receiving phase, every thread receives particle by particle, until a dummy particle is received from every other thread.

The implementation uses two sending/receiving phases. In the first phase, the moving particles are sent to the target thread. In the second phase, the ghost zones are synchronized. We split the algorithm into two phases, because we can only be sure that a thread's ghost zone is up to date when the thread has already received all particles that moved into the thread, since there might be new particles moving into the ghost zone. This results in two barriers per iteration.

In the next subsections, we will show several optimizations.

3.2 2D Partitioning

In order to reduce the size of the ghost zones, we implemented 2D partitioning. As shown in Figure 1, there are now some ghost bins that have to be sent to 3 neighbors. In total, however, the number of bins that have to be sent is smaller. The following expression is an upper bound for the number of bins per thread that must be sent.

$$\frac{2M}{p_x} + \frac{2M}{p_y} = \frac{2M \times (np - p_x) + 2M \times p_x}{p_x \times (np - p_x)} = \frac{2M \times np}{p_x \times (np - p_x)}$$

Since M and np are constants, we can minimize the number number of bins being sent by choosing $p_x = np - p_x = p_y$. Therefore, we should use square processor layouts if possible[1].

3.3 Fixed-sized Send/Receive Buffers

Instead of sending every particle separately, we implemented a send/receive buffer that accumulates particles for a specific thread and only sends them when the buffer is full or the send phase is completed (and there are still some particles in the non-full buffer).

Buffer Size We compiled and ran our implementation several times to find a good buffer size[2]. Smaller buffer sizes lead to a bigger number of sends with only few particle and a lot of overhead for the message passing itself.

Let s be the number of particles in a buffer. The size of the buffer is about $52 \times s$, since every particle is stored in a struct of size 52 bytes.

Figure 2 shows the performance of our algorithm with different buffer sizes. We can see that a larger buffer size improves the performance, because it limits the overhead of passing a lot of messages. In all other benchmarks, we used a buffer size of 630, resulting in a message size of about 32 KB, which is a good message size according to the lecture slides.

[1] This calculation does not account for bins that need to be sent to 3 threads. However, this is only a constant overhead, since it is constrained by the number of threads, and also quite small for a sufficiently large number of bins.

[2] The size of a message is determined by the size of the buffer.

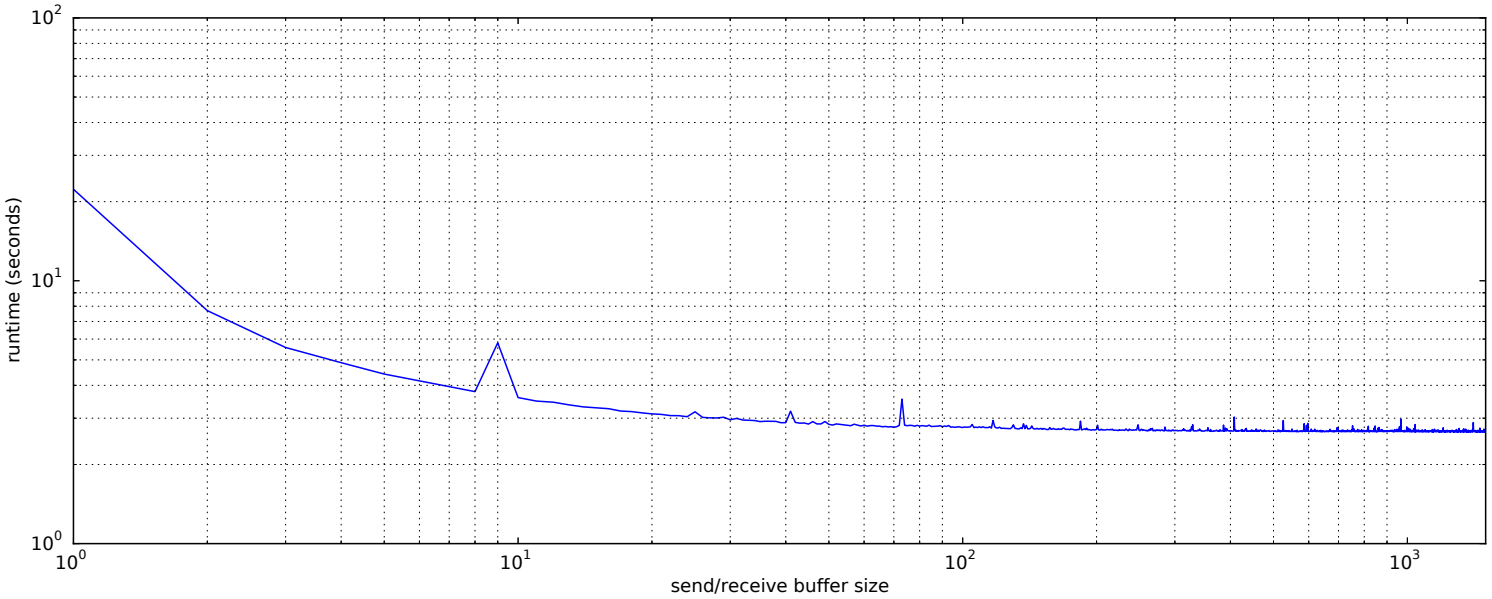


Figure 2: Runtime in seconds of our implementation for $n = 200000$, 1000 iterations, $M = 100$, $p_x = 2$, $p_y = 4$.

Implementation Details Since we are using a buffer instead of sending particles separately, we do no longer need to send a dummy particle to indicate that the last particle was sent. Instead, we send a dummy buffer. In the actual implementation, a thread keeps receiving until a non-full buffer is received. If the buffer size is big, this will almost always be the case for the last buffer. In case the last buffer is full, the sending thread sends a buffer with 0 particles (dummy buffer).

3.4 Limiting communication to neighboring threads

In our first implementation, every thread sent at least one dummy buffer to every other thread, and every thread waited for a message from every other thread, even though some threads do never need to communicate with each other. This kept our implementation simple.

In our final implementation, we added several checks, such that a message is only sent to a thread if this thread is a neighbor. This is based on the assumption that a particle can not jump across multiple bins. Therefore, a particle cannot jump across multiple threads. This optimization required some additional `if` check. However, the overhead for passing additional messages is much higher, if $p_x > 2$ or $p_y > 2$[3].

3.5 Single Send/Receive Phase

Instead of having two send/receive phases for moving particles and ghost particles, as explained in a previous subsection, we decided to merge both phases into a single phase.

When a particle moves to another thread, the sending thread checks additionally whether the bin that the particle moved to is a ghost bin, and which other threads need that ghost bin. Note, that whenever a particle moves to another thread, it in fact always moves to a ghost bin, since all border bins are ghost bins (ghost bins for other threads) and a particle can only move to a neighboring bin. The number of threads that are interested in that ghost bin, can however vary. It could be either 1 or 3 other threads.

In our optimized version, we have only one barrier per iteration (since we have only one send/receive phase). Furthermore, we can use the same buffer for sending moving particles and ghost particles. Therefore, we have the chance to save a message send if otherwise a moving particle buffer and a ghost particle buffer would have been sent, where both buffers together contain less particles than the size of the buffer.

[3]Otherwise, all threads are neighbors and have to communicate with each other, anyway.

3.6 Target Rank Lookup Table for Bins

The implementation of the optimizations presented in the last two subsections turned out to be quite complicated. For every particle, we first need to check, whether the target bin is a ghost bin. Then we need to check which other threads we need to send the particle/buffer to. This involves several special cases, where a thread does not have a neighboring thread, because it is the outermost thread.

In order to reduce the number of branches in the code and to allow more compiler-level optimizations, we maintain a list of thread ranks for every bin. When a particle is moved to a bin, we send it to all threads in that list. This table (bins are arranged in a 2D table) is computed only once at the beginning of the program.

3.7 Memory Scaling

In our original implementation, we maintained an array of all particles per thread. This simplifies[4] memory management because we could calculate the memory position for a particle by adding its tag to the base offset of the array. However, this implementation does not scale well, because it limits the number of particles that can be simulated. When using *np* threads, this approach uses *np* more particles as opposed to our final implementation.

In our final implementation, each thread allocates memory only for the particles that are contained in its bins and for ghost bins. Therefore, we use significantly less memory and the number of particles that can be simulated increases linearly with the number of threads.

We generate all particles on the thread with rank 0. This thread calculates the target thread for every particles and send them to that thread. Afterwards, every thread has only its own particles and the ghost particles.

3.8 Caching and Inlining

We structured our implementation in a modular way, in order to reduce complexity and simplify implementation. For example, sending/receiving moving particles and ghost particles uses the same functions. In fact, bins that belong to a thread and ghost bins are handled and stored in a similar way, except that ghost particles are not taken into account when moving particles and ghost bins are cleared after each iteration. In order to allow for more compiler optimization, we changed our code as follows.

- Add `inline` keywords for functions when possible.
- Cache frequently used values in arrays instead of recomputing them all the time, e.g. the minimum and maximum bin index in x/y direction for every thread, the thread rank that own a bin for every bin, etc.
- Use larger statically allocated arrays of constant size instead of dynamically allocated arrays[5]. For instance, we statically allocate an array of size 3 for every list in the target rank lookup table (see previous subsection), although some bins might be sent to only one or no thread at all.

3.9 Further Ideas

We thought of further ideas to improve the performance of our implementations, that we did not implement. If the number of particles is limited and known at compile time, we can statically allocate memory for all particles on every thread. This allows us to get rid of dynamic memory management and the compiler can directly put the memory locations into the assembly code instead of computing them at runtime. However, this idea does not scale (see section 3.7), as it works only for small numbers of particles.

[4] It also turned out to be faster, because of fewer memory management operations.

[5] This allows the compiler to precompute the position of the array in the memory and to unroll loops.

4 Implementation Details

In this section, we give a high-level overview of the structure of our implementation. The following list is an enumeration of all important steps in our implementation.

1. On rank 0: generate random distribution of particles and send them to other threads.
   - Generate random particles and compute target rank.
   - Send particles to the thread where they belong to (including ghost particles), using `MPI_Recv` and `MPI_ISend`, in groups of up to 630 particles (buffer size).
   - Put particles for rank 0 into the correct bins.
2. On rank > 0: receive particles and put them into the correct bins.
3. `MPI_Barrier`.
4. Setup thread-local variables, fill arrays with cached values, setup target rank lookup table.
5. Simulate a number of iterations.
   (a) Apply forces.
   (b) Clear ghost particle bins (and deallocate memory for ghost particles).
   (c) Move particles.
   (d) Send particles that moved to another thread to that thread and to all other threads that need the particle as a ghost particle, using `MPI_ISend`, in groups of up to 630 particles (buffer size).
   (e) Send local particles that did not change thread ownership (stayed local) to other threads, if they need them as ghost particles. Reuse the same send buffers as in the previous step.
   (f) Flush the send buffers, i.e. send buffers that are not full.
   (g) Receive particles using `MPI_Recv`, in groups of up to 630 particles, from every neighboring thread. A thread will receive only particles that move to that thread or are needed as ghost particles.
   (h) `MPI_Barrier`.
   (i) Reset send buffers.
6. Calculate the error rate locally for particles on the current thread and produce an aggregated result using `MPI_Reduce` on rank 0.

5 Performance Overview

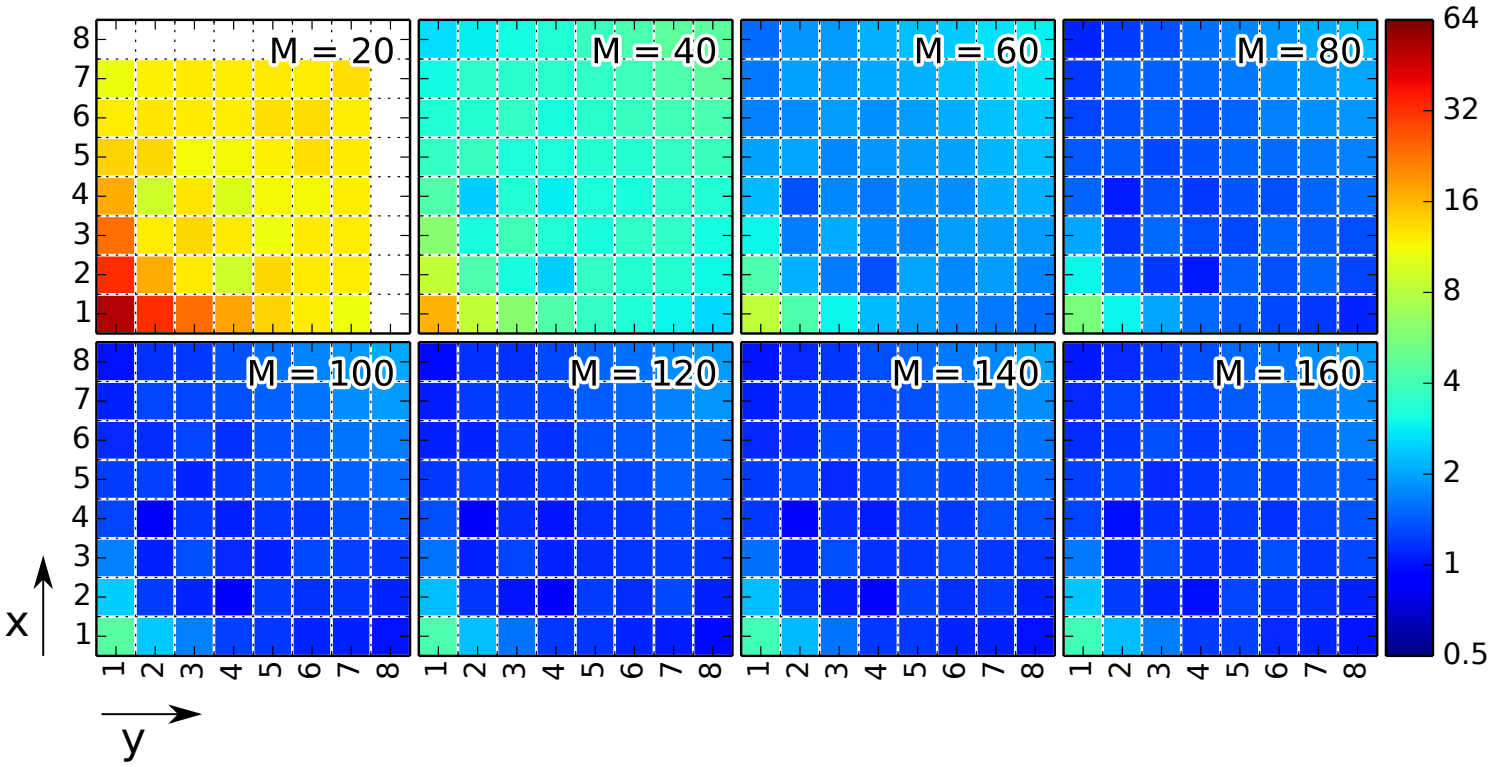


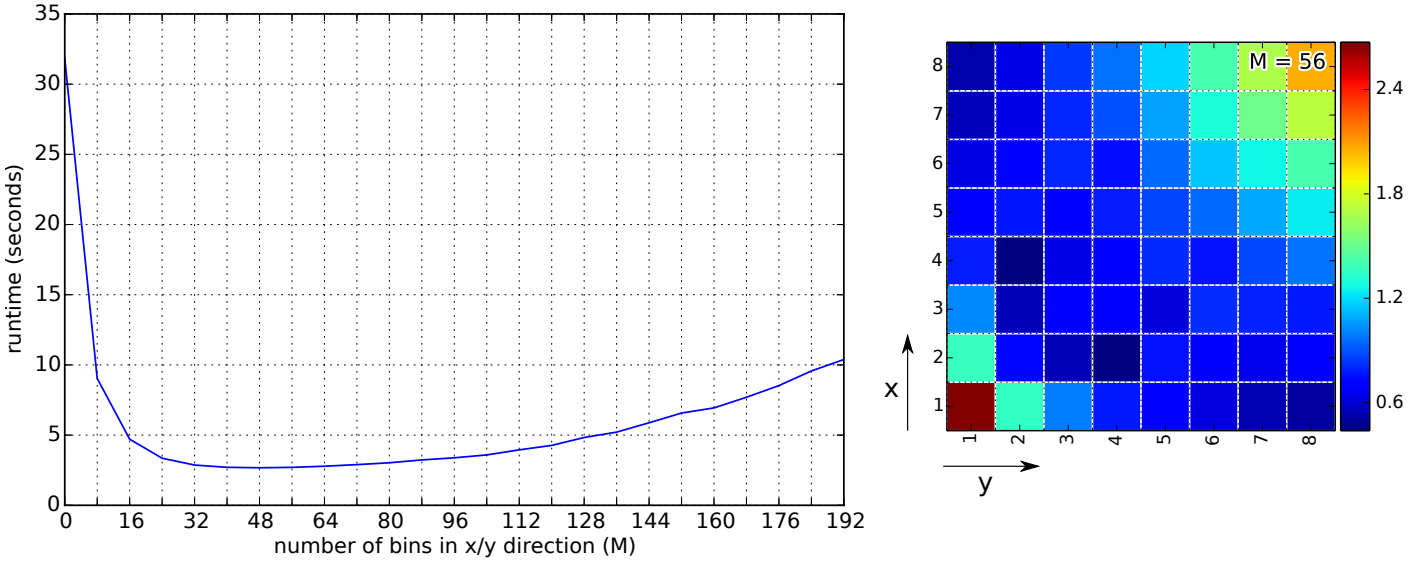
Figure 3: Runtime in seconds of our implementation for $n = 100000$, 1000 iterations, and different values for M at different thread layouts.

Figure 3 gives an overview of the performance of our implementation at different thread and bin configurations[6]. We can see that the performance increases when we increase M, the number of bins. As described in the optimization section, a higher number of bins leads to less particle pairs that need to be examined and to less overhead due to smaller ghost zones.

Furthermore, we can see that, for a fixed value of M, the performance increases with the number of threads, until we use more than 8 threads. At that point we cannot take advantage of more parallelism, because we ran the program on one node with 8 processors only.

[6]For $M = 20$, all configurations with more than 7 threads in one dimension are missing, because we did not have enough time to run them.

6 Strong Scaling Study



(a) Performance at $p_x = p_y = 1$, $n = 8192$ at different values of M. (b) Performance at $M = 56$, $n = 8192$.

Figure 4: Strong performance study, using default parameters.

Benchmarks For every configuration in Figure 4, we ran the program 5 times and took the minimum value. As a seed we used `1359856190`, in order to keep the workload constant. We ran the program with the default parameters, i.e. 1000 iterations. For Figure 4a, we used all multiples of 8 as values for M. The maximum value of M is 200, because the algorithm can otherwise not guarantee that particles will not jump across multiple bins.

Results In Figure 4a, we can see that we reach the peak performance for 1 thread at $M = 56$. In general, a small number of M results in poor performance, because it results in a small number of bins. In that case, bins contain a larger number of particles, i.e. more pairs of particles need to be evaluated for calculating the forces[7].

In addition, a larger number of particles must be transferred to other threads, because the size of the ghost zone grows when the number of bins decreases. For example, for $M = 40$ we have more ghost bins than for $M = 100$, but the bins themselves are smaller, because they are thinner. Since the ghost zone consists of one column/row per border only, a large number of bins causes less particles bins to be transferred.

We can see that for bin sizes greater than 56 degrades. Although the amount of communication decreases further when increasing M further, there are two complementing effects that we need to take into account.

- In every iteration, we send at least one buffer of particles to every neighboring thread (see optimization section). Therefore, we can, at some point, not benefit from less communication anymore, since we have to transfer at least one buffer. Furthermore, at some point, smaller message sizes do not increase the performance significantly anymore, because the overhead for passing the message dominates the time of the operation.

- A bigger number of bins results in more complicated `for` loops and caching data structures. For example, the target rank lookup table for bins grows with the number of bins and certain `for` loops need more iterations when the number of bins grows.

[7] Remember that only the current bin and its neighboring bin are taken into account when calculating forces on particles.

Scaling Figure 4b shows the performance of our implementation at $M = 56$ with different thread configurations. We ran it on one node with 8 processors. When increasing the number of threads, the performance increases until we reach 8 threads. Increasing the number of threads further does not boost the performance, because we there are no more free processors. Our implementation scales almost linearly and is significantly faster than the provided reference implementation that uses OpenMP. We reach the peak performance for the following configurations.

- $p_x = 2$, $p_y = 4$: 0.4388 seconds
- $p_x = 4$, $p_y = 2$: 0.4408 seconds
- $p_x = 1$, $p_y = 8$: 0.5086 seconds
- $p_y = 8$, $p_x = 8$: 0.5325 seconds

These are the configurations where we use exactly 8 threads. Less threads imply less occupancy, more threads slow down the benchmarks, because we have the overhead of setting up and synchronizing threads without benefiting from more processing units. In particular, the we reach the peak performance for 2×4 and 4×2, because these configurations are as close as possible to being quadratic with 8 threads. As we described in the optimizations section, a quadratic layout minimizes the number of particles/bins that have to be transferred.

Parallel Speedup

$$S_p = \frac{\textit{Running time of best serial program on 1 processor}}{\textit{Running time of best parallel program on } P \textit{ processors}}$$

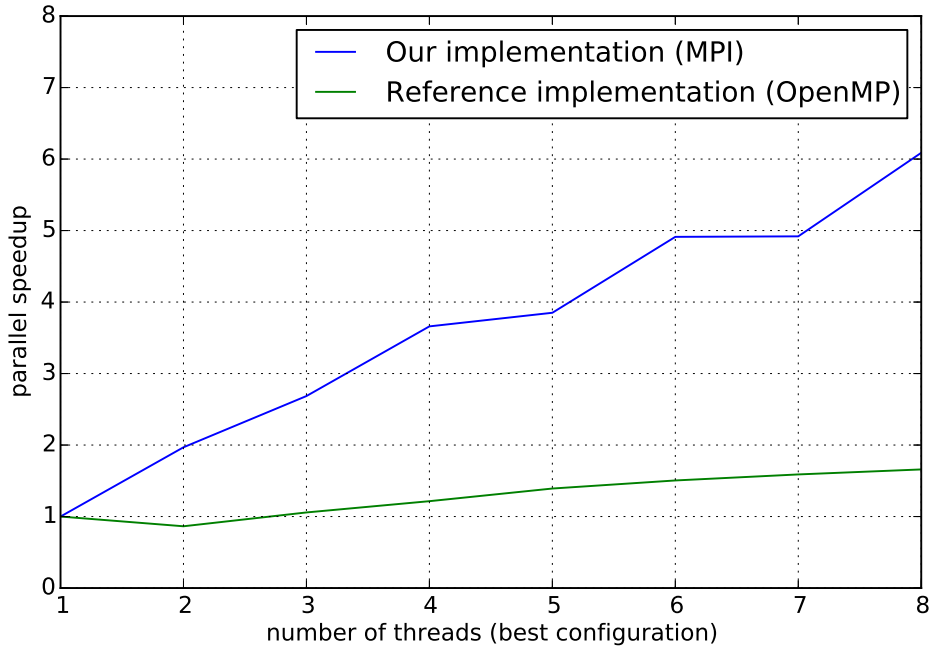


Figure 5: Parallel Speedup of 1 to 8 cores

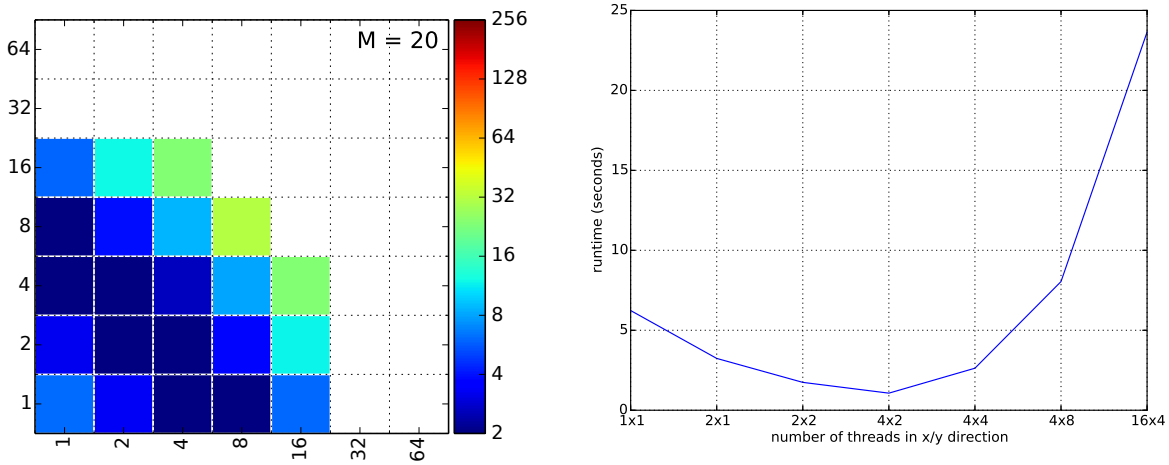
Using the parallel speedup formula, the Parallel Speedup was calculated using 1 to 8 cores. The bin parameter was set to 56 because it was calculated to be the most efficient based on Figure 4, and the other parameters were set to default. An almost linear speedup can be observed in Figure 5. This agrees with what should

theoretically happen. As the number of cores increase for the same work size, the total running time should divide by that number. Because of the overhead of the MPI calls and overhead that is inherent from the program (i.e. for loops, checking particles twice, allocation/deallocation of memory, etc), a perfect parallel speedup was not achieved. The speedup is also limited by the serial section of the program. In this case however, the serial section is only apparent when plotting is enabled and negligible when calculating accuracy (each core calculates its own accuracy measurements which are then sent to core 0 to using `MPI_Reduce`). Finally, because a linear speedup is observed, by definition our program is scalable. Note that the reference design is slower because the bin size parameter used was the same as the one for our program. A case study for the reference program was not done, and hence given another bin size, the reference program could be faster than ours.

7 Weak Scaling Study

For the weak scaling study, we maintained a constant workload per thread (on average) / per node. The idea is that the number of particles in the simulation is proportional to the number of threads / nodes.

7.1 $M = 20$, $n = \#nodes \times 8192$, constant workload per node



(a) Performance at $M = 20$, maintaining a constant workload per node.

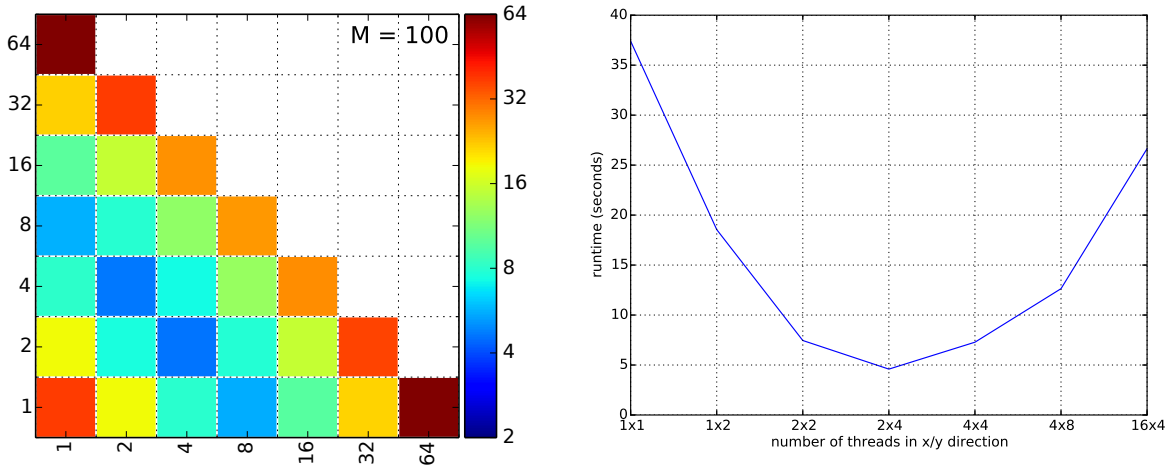
(b) Performance at $M = 20$, increasing thread numbers (best grid configuration).

Figure 6: Weak performance study with $M = 20$.

Figure 6 shows the performance at different thread configurations with a constant workload per node, i.e. we set $n = 8192 \times \lceil \frac{p_x \times p_y}{8} \rceil$. Note, that in Figure 6a, configurations with $p_x > 20$ or $p_y > 20$ are not valid, because this would result in certain threads having no bins at all (so theres no point in that configuration running). We can see, that we reach the peak performance for 8 threads. If we increase the number of threads and use more nodes, we will still get a little bit more speedup, however, the effect of sending particles/buffers via network instead of inter-process communication slows down the program. That is why the performance decreases exponentially, once we start using more and more nodes. Theoretically, a horizontal line should be present instead of a curve because the each node computes the same work, but in parallel so the total time stays the same. Again, we think the inter-network communication is the reason we get a U shaped curve. When the number of threads is less than 8, which means all threads are on one node and no inter-network communication is done, the speed decreases a bit. But once there are more than 8, the running time jumps up, and hence inter-network communication is being done.

Note, that if we run the same workload on more processors, it will still be faster as long as we choose a good processor grid. In the next section, we will show the performance for a bigger number of bins and particles.

7.2 $M = 100$, $n = \#nodes \times 65536$, constant workload per node



(a) Performance at $M = 100$, maintaining a constant workload per node.

(b) Performance at $M = 100$, increasing thread numbers (best grid configuration).

Figure 7: Weak performance study with $M = 100$.

Figure 7 shows a weak scaling study for more particles and a greater number of bins. If we increase the number of threads to more than 8 threads, the performance does still not scale linearly. However, the factor of the logarithmic scaling is smaller and running the same workload on multiple nodes is still significantly faster than running it on a single node.

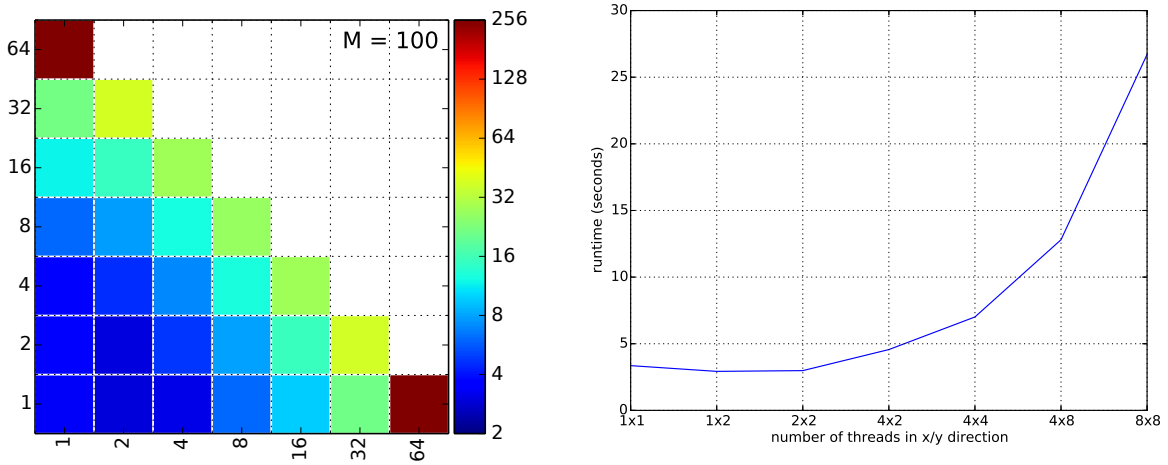
- As long as we stay on a single node, the overhead for passing messages is dominated by inter-process communication.
- When we start using more than one node, the overhead for passing messages is dominated by network communication, which is much slower and has a much higher latency. A higher buffer size might be beneficial, however, we did not run further benchmarks.
- When using running the simulation with more particles, the amount of time spent on the actual computation increases, while the time spent on network traffic does not increase in the same way when we also increase the number of bins. Therefore, we get a better speedup in Figure 7 than in Figure 6.
- We wanted to use the MPI profiler, but did not have enough time to run jobs on Carver so unfortunately we can not be 100% sure that the MPI calls are dominating the run time. We also don't know if the bins are allocated Round Robin, i.e. thread 0 node 0, thread 0 node 1, thread 0 node 2,.. If it is allocated as this order where each thread per node is allocated bins of increasing order, then on one node, there will be bins that are very far from each other. If this is happening, performance will take a serious hit because the probability of having to do inter-network communication increases dramatically since there are no two threads in a node that will send ghost bins to each other.

7.3 Conclusion

For the weak scaling study, we reach the best performance when running on 8 threads, because the amount of computation per node is constant, we use all processors on one node, and there is no overhead for sending messages via network.

Appendix

Weak Scaling: $M = 100$, $n = p_x \times p_y \times 8192$, constant workload per thread

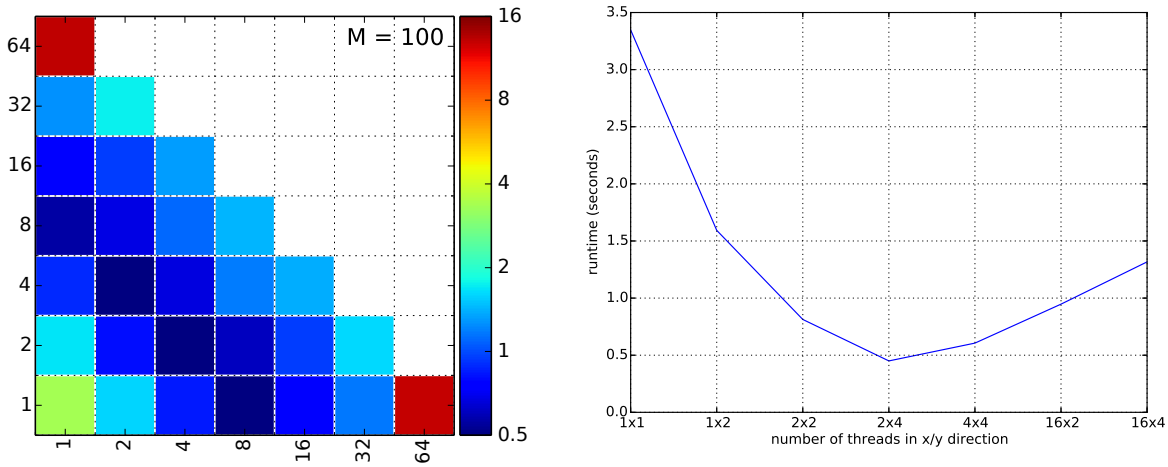


(a) Performance at $M = 100$, maintaining a constant workload per thread.

(b) Performance at $M = 100$, increasing thread numbers (best grid configuration).

Figure 8: Weak performance study with $M = 100$.

Weak Scaling: $M = 100$, $n = \#nodes \times 8192$, constant workload per node



(a) Performance at $M = 100$, maintaining a constant workload per node.

(b) Performance at $M = 20$, increasing thread numbers (best grid configuration).

Figure 9: Weak performance study with $M = 100$.