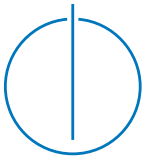
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementation and Optimization of the Midpoint Method in ls1-mardyn

Sascha Sauermann



DEPARTMENT OF INFORMATICS

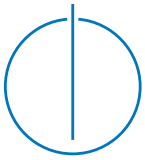
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementation and Optimization of the Midpoint Method in ls1-mardyn

Implementierung und Optimierung der Midpoint Methode in ls1-mardyn

Author:	Sascha Sauermann
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisors:	M.Sc. (Hons) Steffen Seckler
	M.Sc. Nikola Tchipev
Submission Date:	15.09.2017



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.09.2017 Sascha Sauermann

Acknowledgments

I want to thank Steffen Seckler for answering all my questions, helping me to understand the code of MarDyn and providing valuable feedback and corrections. Additionally, I want to thank Nikola Tchipev for introducing me to the topic of Molecular Dynamics in the course "PSE Molekulardynamik".

Abstract

Simulations are an important tool in Molecular Dynamics that often substitutes costly physical experiments. Because of the high number of particles and the small time resolution, a massive computational effort is required, which makes this also an interesting topic in the area of high-performance computing. As standard methods for the computation of particle interactions like the Full Shell method do not scale that well with the large number of processors available in supercomputers today, new approaches are needed. The Midpoint Method is such a highly scalable method that requires less communication between processes, allows deeper parallelization and even improves sequential performance for dense scenarios. It was implemented into MarDyn, a state of the art framework for Molecular Dynamics simulations, and optimized for performance. Benchmarks show that the Midpoint Method performs better than Full Shell and Half Shell in many cases, especially in dense scenarios.

Contents

1. Introduction

Molecular dynamics simulations are a valuable tool in many different areas, like biology, physics, chemistry or material science, as they are a cost efficient replacement for traditional experiments and allow computation of properties like pressure or temperature which are difficult to measure on such a small scale.

The computational effort of such simulations is enormous, as the time resolution typically lies in the range of femtoseconds while simulating times up to multiple microseconds [4]. Particle numbers can reach as high as multiple trillions (10^{12}) [6]. Molecular dynamics is therefore also an interesting topic for high-performance computing.

To have acceptable run times, massive parallelization is necessary, but classic methods like the Full Shell method scale only up to a point. Hence, new methods that scale better and allow computations with more processes are needed.

The Midpoint method, as described by J. Bowers [1], is such a highly scalable approach. It requires less communication between processes and allows a parallelization with up to eight times more processes than the classic methods.

To test the performance of this method it was implemented into MarDyn, a fast and highly scalable framework for Molecular Dynamics simulations [6], and some performance optimizations were performed. Benchmark comparisons with the Full Shell and Half Shell methods were performed and showed that the Midpoint Method is especially advantageous in dense scenarios with many particles per cell. Only in very sparse domains, it is outperformed by the other two.

This paper starts with an introduction to Molecular Dynamics including an explanation of the Full Shell method, Half Shell method and Midpoint Method. Then we describe some details of the implementation and finish with the benchmark.

2. Theory

2.1. Molecular Dynamics

The simulation of particles is based on Newton's equations of motion, second order ODEs. Simulating single particles has the advantage that interactions between them like collisions can be observed. Additionally, macroscopic values like energy, temperature or pressure can be calculated and compared over time.

For a Molecular Dynamics simulation there are three core components [4]:

- **Appropriate interaction potential**
  Depending on what particles should be simulated, a potential has to be found that represents the real world. One could consider gravitation as a simple example.

- **Time integration**
  To calculate a solution of the equations of motion, we discretize time and only evaluate the equations at those discrete points. In every time step, the forces acting on each particle have to be evaluated.

- **Fast force evaluation**
  As the trivial approach to compute the forces between N particles – a particle interacts with every other particle – has a complexity of $\mathcal{O}(N^2)$, more advanced methods are required.

We will delve into these three aspects in the subsequent sections.

2.1.1. Lennard-Jones Potential

The potential commonly used for short-range interactions is the 12-6 Lennard-Jones potential. It models a repulsion based on the *Pauli repulsion* and an attraction based on the *Van der Waals forces*. The potential is parameterized by two parameters:
(1) The depth of the potential well ε that defines the strength of the forces and (2) the zero crossing σ which defines the distance between two particles where the attracting force is equal to the repulsive one. Experimental data would be used to fit those parameters.

This LJ potential is defined [4] as

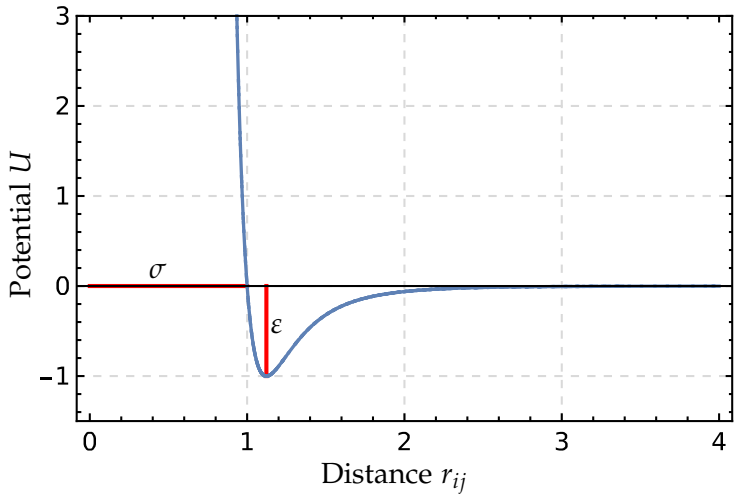


Figure 2.1.: The Lennard-Jones Potential for $\varepsilon = \sigma = 1$. The zero crossing is at $x = \sigma$ and the lowest point is inside the so called potential well at $y = -\varepsilon$.

$$U\left(r_{ij}\right) = 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] \tag{2.1}$$

where $r_{ij} = \|x_j - x_i\|_2$ is the distance between the positions x_i and x_j of two particles i and j. The force F_{ij} a particle j applies to a particle i is defined by the negative gradient of the potential with respect to the position.

$$F_{ij} = -\nabla_{x_i} U\left(r_{ij}\right) = -\frac{24\varepsilon}{r_{ij}}\left[\left(\frac{\sigma}{r_{ij}}\right)^{6} - 2\left(\frac{\sigma}{r_{ij}}\right)^{12}\right] \tag{2.2}$$

To now get all the forces acting on one particle i, we sum up all forces applied to i. This sum excludes F_{ii}, because a particle does not apply a force to itself.

$$F_i = \sum_{\substack{j=1 \\ j\neq i}}^{N} F_{ij} \tag{2.3}$$

2.1.2. Störmer-Verlet Method

As Newton's equations of motion are a second order ODE in the position, we discretize them in time to calculate a solution. This results in multiple time steps of size δt which we number with 0 to n. In each step, we calculate the new particle position x_i^{n+1} and velocity v_i^{n+1} from the old position x_i^n and velocity v_i^n under consideration of the mass m_i and the current force F_i^n. See chapter 3.1 of reference [4] for a derivation of the following formulas.

The Störmer-Verlet Method is defined as:

$$x_i^{n+1} = 2x_i^n - x_i^{n-1} + (\delta t)^2 \frac{F_i^n}{m_i} \tag{2.4}$$

This method has some disadvantages though. As the force term of the sum is very small because it depends on the square of the time step but the position may be large, significant rounding errors might occur. The second problem is that we never calculated the velocity which is needed for macroscopic values like the kinetic energy. However, they can be approximated as the central difference of the positions:

$$v_i^n = \frac{x_i^{n+1} - x_i^{n-1}}{2\delta t} \tag{2.5}$$

Two variants of this method that are less susceptible to rounding errors exist:

- **Leapfrog scheme**
  For the Leapfrog scheme, the velocities are calculated at $t + \delta t/2$ like

  $$v_i^{n+1/2} = v_i^{n-1/2} + \frac{\delta t}{m_i} F_i^n \tag{2.6}$$

  then the positions can be determined as such:

  $$x_i^{n+1} = x_i^n + \delta t \cdot v_i^{n+1/2} \tag{2.7}$$

  As the velocities are calculated at another point in time as the positions, they have to be averaged to get the values at time t.

  $$v_i^n = \frac{v_i^{n+1/2} + v_i^{n-1/2}}{2} \tag{2.8}$$

- **Velocity-Störmer-Verlet Method**
  Another variant is the Velocity-Störmer-Verlet Method that calculates the position as

  $$x_i^{n+1} = x_i^n + \delta t \cdot v_i^n + \frac{F_i^n \cdot (\delta t)^2}{2m_i} \tag{2.9}$$

  and the velocities at the same point in time like:

  $$v_i^{n+1} = v_i^n + \delta t \frac{F_i^n + F_i^{n+1}}{2m_i} \tag{2.10}$$

Both need about the same amount of memory and have an error of order two [4].

2.1.3. Linked Cells Method

The trivial approach to calculating the forces is to consider each particle pair. This results in $\mathcal{O}(N^2)$ calculations for N particles in the system.

As the LJ potential decays very fast with larger distances (see Figure 2.1), there is a distance r_c so that all forces from particles with a distance bigger than r_c can be neglected without a significant error.

We therefore only calculate forces with particles that are inside (or on) such a sphere around a particle i. The distance r_c is thus called cutoff radius and is typically chosen as 2.5σ for the LJ potential.

However, now we have to calculate the distances between all particle pairs to decide which forces to compute. This still has a complexity of $\mathcal{O}(n^2)$.

To reduce the number of pairs that have to be considered in each step, we subdivide our simulation domain into small cells of size $r_c \times r_c \times r_c$. This limits possible pairs to the 26 neighboring cells — including diagonals, or 8 neighbors in 2D (see Figure 2.2).

This reduces the complexity to $\mathcal{O}(n)$ as we only have to check a small final number of possible partners for each particle – all particles in the neighboring cells.

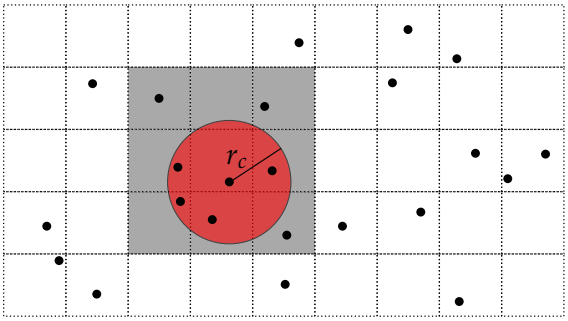


Figure 2.2.: Linked Cells Method: Domain subdivided into cells; Cutoff radius around a particle (red) and the 8 neighbor cells (gray) to the cell containing this particle. We only have to check the gray cells to find all particles that possible lie within the cutoff radius.

We can index all cells so that they can be stored in a 1D array. Each cell then contains a list of the particles within it. As particles can move out of a cell, the particle list of each cell has to be rebuilt every time step.

2.1.4. Boundary Conditions

The domain of our simulation is finite. This results in the question what to do if a particle leaves this domain. There are three typical solutions to the problem:

1. Outflow: Delete particles that leave the domain

2. Reflection: Prevent particles from leaving by reflecting them

3. Periodic boundaries: Reinsert particles on the opposite side

We will focus on the periodic boundaries as they are the most commonly used ones. We get those by sticking opposite sides of our domain together. In 2D this would correspond to creating a torus from our plane, so that opposite sides of our domain are adjacent.

To implement this with our Linked Cells method, we add another layer of cells around our existing domain and call them the halo cells. The cells adjacent to the halo cells are called boundary cells. When now a particle enters a halo cell – a so called *Leaving particle*, it is shifted across the domain and inserted into the opposite boundary cell. If the particle leaves the domain at a corner (or an edge in 3D) multiple shifts are necessary.

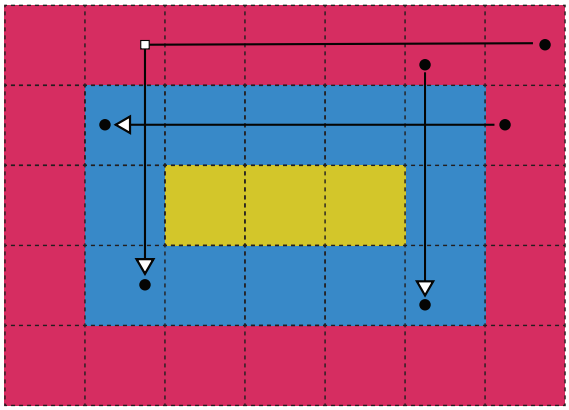


Figure 2.3.: Inner cells (yellow), boundary cells (blue) and a ring of halo cells (red) around them. Leaving particles are shifted 3 or 5 cells to the opposite side of the domain. The particle that moved into the corner halo cell requires one shift per direction to be positioned in a boundary cell.

This solves the reinsertion of particles but as two opposite boundary cells should be neighbors forces between particles in them have to be calculated. The simplest way to do this is to copy all particles from boundary cells into the opposite halo cells (for corners or edges multiple copies are required), interact boundary and halo cells and then delete the particles in the halo cells again. We call those fake particles the *Halo copies*.

2.1.5. Simulation Loop

We can now combine all this into the main simulation loop. In each iteration the following steps have to be executed:

1. **Calculate new positions** of all particles with some variant of the Störmer-Verlet Method (see section 2.1.2).

2. **Apply boundary conditions** and shift the leaving particles to the opposite side of the domain and create halo copies (see section 2.1.4)

3. **Rebuild linked cells** and insert particles into each cell (see section 2.1.3).
4. **Calculate forces** by using some cell iteration method to interact specific cells (See section 2.1.3 for a simple approach or section 2.3, 2.4 and 2.5 for more sophisticated ones). Use the LJ potential (see section 2.1.1) to compute forces between two particles.
5. **Calculate the new velocities** depending on the method used for calculating the positions.
6. Optional: Perform measurements, write output files, ...
7. **Increment current time** by adding the time step

We can now compute everything necessary, but we are quite slow. The first step to gain performance is to parallelize the computations by splitting the domain (see section 2.2). The next step is to improve the cell iteration method which will be described in the sections 2.3, 2.4 and 2.5.

2.2. Parallelization

2.2.1. Domain Decomposition

To parallelize the Linked Cells method for n processes, at first the domain is split into n smaller cuboids, and a shell of halo cells around each one is added. The outer-most non-halo cells are the new boundary cells for this process. Figure 2.4 shows an example how the domain is split for four processes.

The simplest form of splitting the domain is a regular grid of same sized cuboids. However, this might result in a suboptimal load distribution as some cuboids may contain many particles and others just a few. Other domain decomposition methods solve this problem. When using a k-d tree to create cuboids, the unbalance between the cuboids can be minimized.

Each cuboid is the part of the domain assigned to a single process. This process is responsible for all particles in the cuboid – known as spatial decomposition – and performs the normal simulation loop described in section 2.1.5 with some changes.

First of all the periodic boundary conditions have to be adapted as we now also have boundaries between each process:

- **Export/Import leaving particles**
  Particles that are outside of a process' domain (moved into a halo cell) have to be sent to the process that now owns this particle, and removed from the previous process.
- **Export/Import halo copies**
  To calculate interactions between neighboring cells that were separated into two

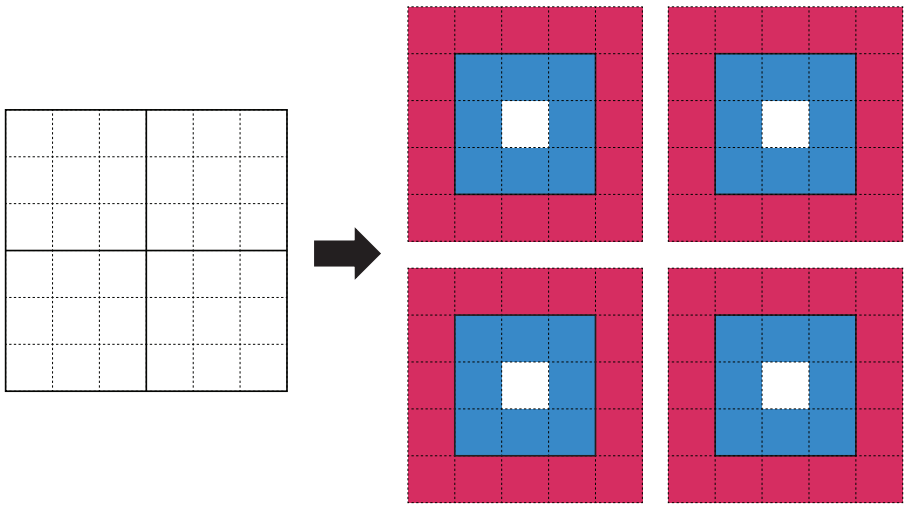


Figure 2.4.: Domain decomposition of a domain of 9×9 cells with 4 processes. The domain is split into four 3×3 squares. The newly added halo cells are colored red, and the cells that are now boundary cells are blue.

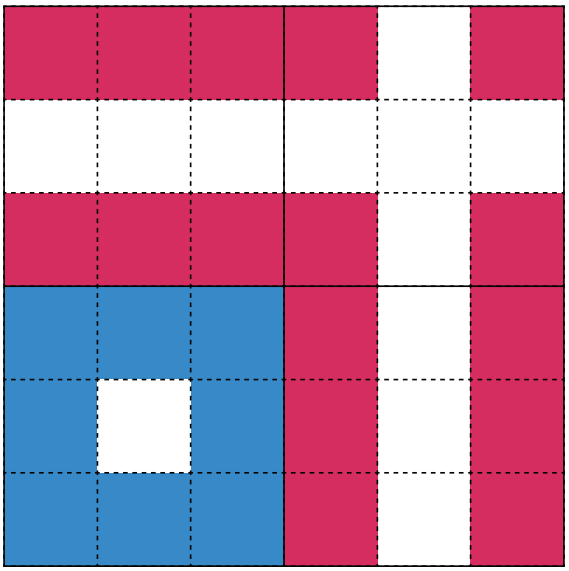


Figure 2.5.: Domain decomposition of a domain of 9×9 cells with 4 processes. The coloring is for the process in the bottom left corner. Boundary cells are blue, and cells that are imported into the halo cells are red.

different cuboids, each process has to import the cells it is missing from its neighbors.

Each process sends all particles from boundary cells to all neighbors that require those cells and places received particles into the corresponding halo cells. Figure 2.5 shows the location of cells that have to be imported for a process.

Sending the boundary cells is sufficient, but we can send even fewer particles as described in section 2.2.2.

Additionally, it might be required to export or import forces across different processes as explained in section 2.2.3.

2.2.2. Import Volume

The reason we import particles is to interact them with particles within our domain. As we only compute forces between particles with a distance smaller than r_c, we do not need to import the ones that are further away from the edges of our domain. This results in an import region that is a (hollow) cuboid with rounded corners and edges (see Figure 2.6 for a 2D example).

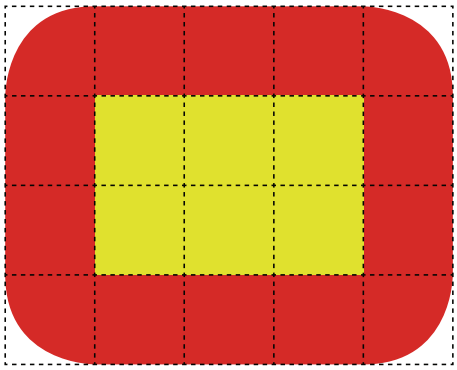


Figure 2.6.: Import region (red) of a process with the size $3r_c \times 2r_c$ (yellow). The region is everything with a distance of up to r_c to the edge of the process' domain.

We now want to calculate the volume of the import region around a process of size $b_x \times b_y \times b_z$. We can split the import region into a sphere consisting of the 8 corners with a radius of r_c and a volume of

$$V_s = V_{\text{sphere}} = \frac{4}{3}\pi r_c^3 \tag{2.11}$$

and three cylinders consisting of four edges each with a radius of r_c and a height of b_d where $d \in \{x, y, z\}$. They have a volume of

$$V_c^d = V_{\text{cylinder},d} = \pi r_c^2 \cdot b_d. \tag{2.12}$$

The remaining parts are three cuboids where one side has the length $2r_c$ and the other two the length b_u and b_v where $(u,v) \in \{(x,y),(x,z),(y,z)\}$. They have a volume of

$$V_q^{u,v} = V_{\text{cuboid},u,v} = 2r_c \cdot b_u \cdot b_v. \tag{2.13}$$

Thus, we get the import volume as

$$V_i = V_s + V_c^x + V_c^y + V_c^z + V_q^{x,y} + V_q^{x,z} + V_q^{y,z} \tag{2.14}$$

$$= \frac{1}{3} r_c (3b_y(\pi r_c + 2b_z) + 3b_x(\pi r_c + 2b_y + 2b_z) + \pi r_c(4r_c + 3b_z)) \tag{2.15}$$

or if our process is cubic with $b = b_x = b_y = b_z$ as

$$V_i = 6b^2 r_c + 3b\pi r_c^2 + \frac{4}{3}\pi r_c^3. \tag{2.16}$$

We will assume that each process has a cubic domain from now on.

2.2.3. Force Exchange

As we will see in section 2.4 and 2.5, not all spatial decomposition methods have their processes calculate every interaction for the particles they are responsible for. To combine the forces from all those processes they have to be imported from the neighbors. This means every process has to export the forces of some halo particles. These are the same halo copies that were previously imported [1]. This means the import region of the halo copies is the same region used for exporting the forces (e.g. the red area in Figure 2.6).

2.2.4. Overlapping

Instead of waiting until the communication is completed and then continuing the calculations, both can be overlapped. As only the boundary cells depend on imported particles, such as leaving particles and halo copies, the interactions between inner cells can already be computed while waiting for the communication to finish. Do so can improve the performance.

2.3. Full Shell

In section 2.1.3 we described the Linked Cells algorithm and stated that a cell has to interact with all its 26 neighbor cells. This interaction scheme is known as Full Shell (FS) because a cell interacts with all the adjacent cells – the shell.

By applying this method, we calculate the forces between two particles i and j twice (or never). Once when processing the cell c_i of particle i and calculating the interaction $i \leftarrow j$ to get the force F_{ij} of j acting on i, and once like $j \leftarrow i$ to get F_{ji} when processing c_j.

Based on Newton's third law – Force equals counter force – those two interactions can be combined to one interaction $i \leftrightarrow j$ where $F_{ji} = -F_{ij}$. Hence, we can remove nearly

half of the computations – all except the multiple interactions between boundary and halo cells (see section 2.4 on how to remove those).

We can ensure that each particle pair is processed only once by splitting the neighbors of each cell into 13 forward and 13 backward neighbors. Each cell c then processes only particle pairs $i \leftrightarrow j$ where i is inside c and j is either also in c or in a forward neighbor cell (see Figure 2.7). The missing calculations are done when the backward neighbor cells are processed and c is a forward neighbor of them.

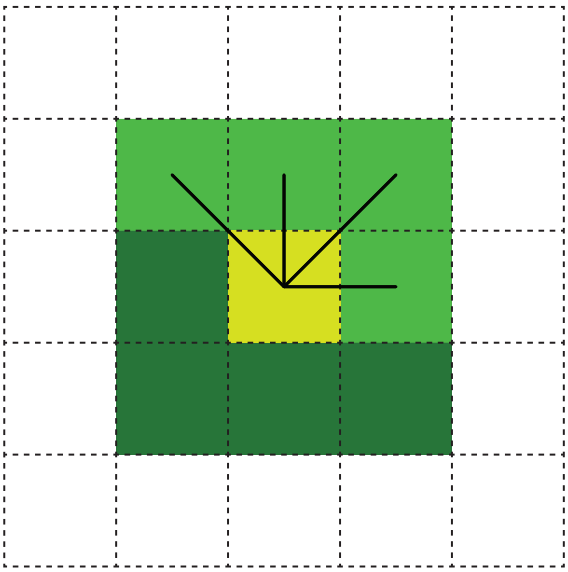


Figure 2.7.: The four forward neighbors (light green) and the four backward neighbors (dark green) of the yellow cell in 2D. Only the four drawn cell interactions with the forward neighbors are computed when processing the yellow cell.

This is sometimes already called Half Shell because most of the cells only process interactions with 13 neighbors instead of 26, but we will call this cell processing method Full Shell as we still have the complete shell as import region and therefore still the same import volume V_i^{FS} as defined in equation 2.16.

2.4. Half Shell

2.4.1. General Idea

In the previous section we reduced the number of cell interactions most cells have to process from 26 to 13 by using Newton's third law. This excluded interactions between boundary and halo cells though.

The reasons for this are the halo copies i' and j' of the two boundary particles i and j introduced in section 2.1.4. We calculate the interaction $i \leftrightarrow j'$ in the boundary cell c_i and $j \leftrightarrow i'$ in c_j. We can not simply remove one of both interactions.

The Half Shell (HS) method further improves Full Shell by removing one of both interactions and replacing it by a force exchange where the force calculated on the halo copy is sent to the original as described in section 2.2.3. Each cell only interacts with the forward neighbors as shown in Figure 2.8.

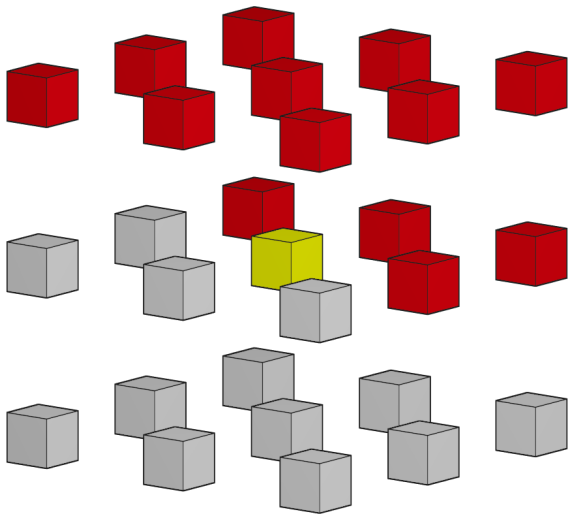


Figure 2.8.: Half Shell interactions: The yellow (currently processed) center cell interacts with each of the 13 red cell (forward neighbors), using Newton's third law to calculate the opposing forces. The interactions between the yellow and the gray cells are calculated when those are processed as the yellow cell lies within the forward neighbors of each of the gray ones.

Each particle interaction is therefore calculated in one of the two cells both particles reside in, which is called traditional spatial decomposition [2].

2.4.2. Advantages

The Half Shell method presented here removes calculations in favor of a force exchange. Depending on the scenario, the performance compared to Full Shell can vary. It is a trade-off between the time computation and communication takes.

The computational effort between boundaries and halos is halved by removing one of two interactions.

The communication is increased by not only having to send the halo copies but also to receive the calculated forces for each of them. However, as we only interact with halo cells that are forward neighbors, we can stop importing halo copies from neighboring processes in the backward direction. This halves our import volume and reduces the required communication effort. Figure 2.9 shows an example for the different import regions of Full Shell and Half Shell.

$$V_i^{\mathrm{HS}} = \frac{1}{2} V_i^{\mathrm{FS}} = 3b^2 r_c + \frac{3}{2} b\pi r_c^2 + \frac{2}{3}\pi r_c^3 \tag{2.17}$$

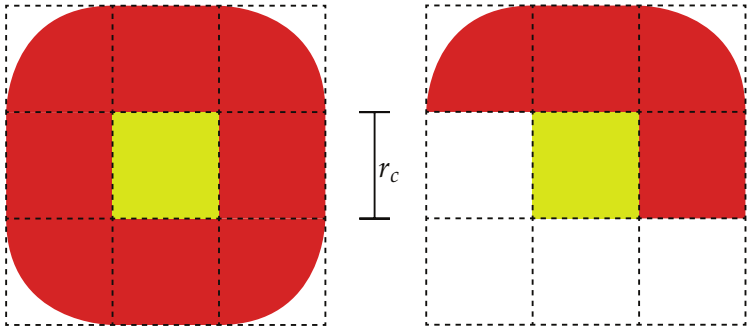


Figure 2.9.: Import region (red) of 2D FS (a) and HS (b) for the minimal possible process size of $r_c \times r_c$.

We expect Half Shell to outperform Full Shell in most cases, but for deep parallelization levels, the communication overhead can be too great.

2.5. Midpoint Method

2.5.1. General Idea

In the Midpoint Method, each cell is responsible for a set of particles regardless where the interactions with these particles are computed. As the name suggests, the midpoint of two or more particles is used to determine which cell interacts them [1]. We will focus on interactions of two particles.

2.5.2. Simplification to Cell Pairs

The calculation of a midpoint $\frac{x_i+x_j}{2}$ is not a cheap operation, as it requires a vector addition and a multiplication with a scalar. Additionally, the cell that actually contains the calculated midpoint must be found.

A simpler method to find all particle pairs with a midpoint in some cell O is to process the following 63 cell interactions while considering Newton's third law [7]:

- All particle pairs inside O itself (1 pair)
- Forward neighbors, as used for Half Shell, with O (see Figure 2.8) (13 pairs)
- Opposite corners of the cube around O (see Figure 2.10) (4 pairs)
- The center cell of three sides with the opposite sides (see Figure 2.11) (3×9 pairs)
- The middle cell of six edges with the opposite edge (see Figure 2.12) (6×3 pairs)

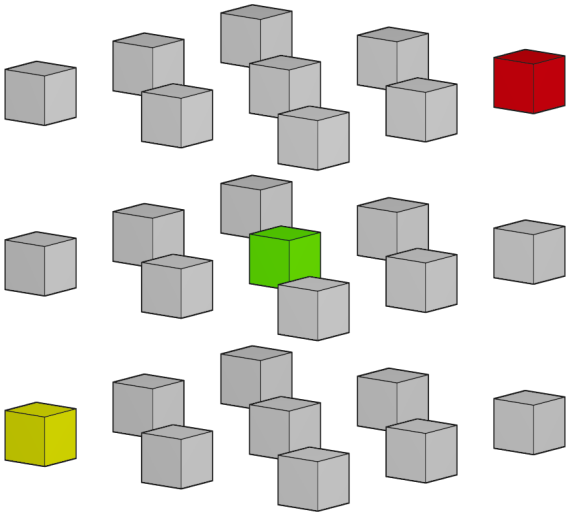


Figure 2.10.: Midpoint interactions (1) - Opposite corners: While the green origin cell is processed, the interactions between opposite corner cells are calculated. This results in four cell pairs like the yellow and red one.

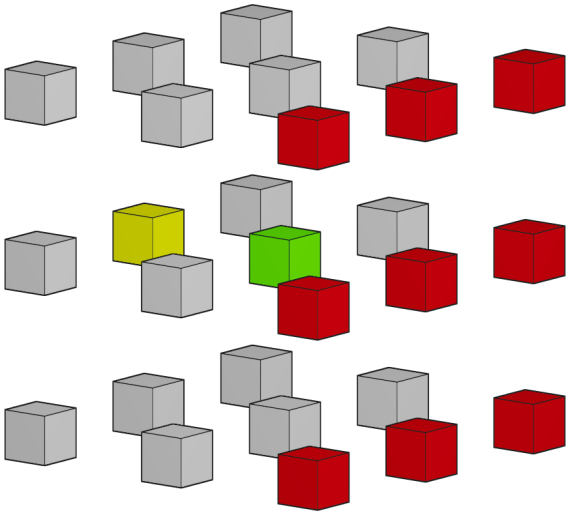


Figure 2.11.: Midpoint interactions (2) - Opposing side: The yellow cell is the center cell of one of the cube's sides and the red cells form the opposite side. The yellow cell is interacted with each red cell while the green origin cell is processed. This is done once per dimension so that no side is used multiple times.

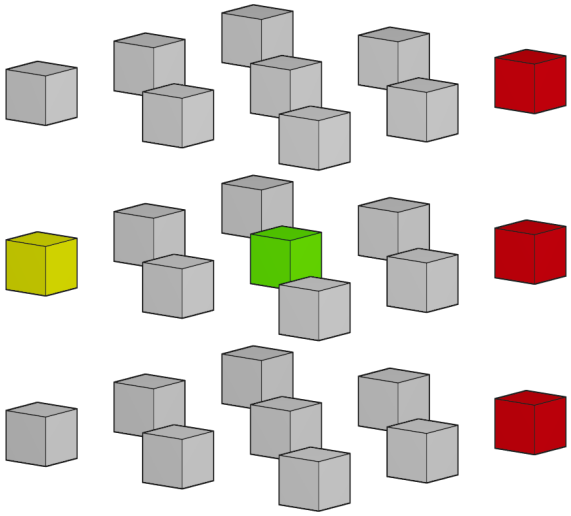


Figure 2.12.: Midpoint interactions (3) - Opposing edge: The yellow cell is the middle cell of an edge and the red cells form the opposite edge. While the green cell is processed the yellow cell is interacted with each red cell. This has to be done for six different edge pairs so that no edge is used twice.

2.5.3. Advantages

Using the Midpoint Method allows to considerably reduce the import region for each process compared to Full Shell or Half Shell as we only have to consider particles with a distance of $\frac{r_c}{2}$ into each direction. Particle pairs where one particle resides in the boundary cells of the process and the other outside, more than $\frac{r_c}{2}$ away, have their midpoint by definition outside of the process' domain [1]. Hence, the neighboring process handles those.

The import region is like the one of Full Shell, a (hollow) rounded cuboid that only depends on the dimensions of the process' domain; but this time only with a distance of $\frac{r_c}{2}$ [1]. The volume can be calculated the same way as in section 2.2.2, just with half the cutoff radius. This results in a volume of

$$V_i^{\mathrm{MP}} = 3b^2 r_c + \frac{3}{4} b \pi r_c^2 + \frac{1}{6} \pi r_c^3. \tag{2.18}$$

We can now compare the import volume of the Midpoint Method with Half Shell and Full Shell by calculating the quotient:

$$\frac{V_i^{\mathrm{MP}}}{V_i^{\mathrm{HS}}} = \frac{36b^2 + 9\pi b r_c + 2\pi r_c^2}{36b^2 + 18\pi b r_c + 8\pi r_c^2} \tag{2.19}$$

This shows that the Midpoint Method always has a smaller import region than Half Shell, as the quotient is always smaller than one for reasonable values of $b > 0$ and $r > 0$. Table 2.1 shows the relative differences for some cutoff radii as box size. As we

b	$5r_c$	$4r_c$	$3r_c$	$2r_c$	r_c	$r_c/2$
$\sim V_i^{\mathrm{MP}}$	$87.3r_c^3$	$57.9r_c^3$	$34.6r_c^3$	$17.2r_c^3$	$5.9r_c^3$	$2.5r_c^3$
$1 - V_i^{\mathrm{MP}}/V_i^{\mathrm{HS}}$	13.26%	15.95%	19.98%	26.72%	40.04%	
$1 - V_i^{\mathrm{MP}}/V_i^{\mathrm{FS}}$	56.63%	57.97%	59.99%	63.36%	70.02%	

Table 2.1.: Absolute volume of MP and relative difference between the import volume of MP to HS, and MP to FS for different box sizes.

can see the smaller the box size the smaller is the import region of the Midpoint Method relative to Half Shell. This is also the case for absolute values.

The import volume of the Midpoint Method has the same size as the import volume of the Eight Shell method but it is larger than the import of other Neutral Territory methods for high processor counts [1].

It is a bad idea though to completely import all cells that intersect the region, as the advantage of the smaller import volume would be lost again for a cell size of r_c. The same 26 neighbor cells as for Full Shell would be imported and even particles we do not need would be transmitted. Because of this we have to make sure that we only import particles within this region independently of the cell structure. For easier computation if a particle lies within the region, sharp corners are used instead of the rounded ones described above, which results in a minimal larger import volume.

However, we can further improve the minimal possible import volume by reducing the cell size to $\frac{r_c}{2}$ as this allows us to choose a box size of $\frac{r_c}{2}$. The import volume for a box of this size can be found in table 2.1 and is even 58.30% smaller than the volume for r_c.

We could now simply import the 26 neighbor cells and still keep this benefit. We increase the number of cell iterations though, as we now have 8 times more cells, and lose some performance in this regard.

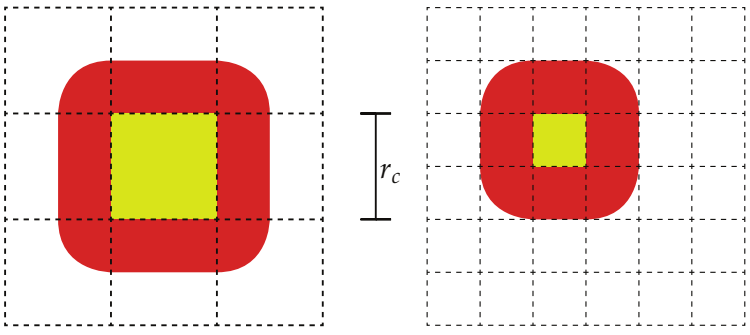


Figure 2.13.: Import region (red) for a minimal sized process of 2D Midpoint: (a) with a cell size of r_c and (b) a cell size of $\frac{r_c}{2}$.

By halving the cutoff radius r_c, we additionally gain the following:

- **Deeper parallelization possible**
  As we have 8 times more cells, we can utilize up to 8 times more processors and therefore reach higher performance in high-density scenarios where the gain from deeper parallelization outweighs the additionally communication.
- **Reduced communication bandwidth**
  As described above we can get to one-eighth of the import volume of Full Shell while still only having to import the neighbor cells directly (or diagonally) adjacent to the process. Fewer particles imported reduce the required communication bandwidth and therefore the time if the available bandwidth was exceeded.
- **Fewer cutoff radius checks**
  As interacted cells still contain particles whose distance is larger than the cutoff radius, each pair has to be checked if the forces should be calculated. Smaller cells reduce the number of needlessly iterated and checked particle pairs. The performance gain for this is small but especially visible at shallow parallelization levels.
- **Better vector register utilization**
  Another consequence of fewer needlessly checked particle pairs is that we have fewer cache misses when performing vectorization within the force calculation. As the transfer time of data to a register depends on the cache, cache misses are costly and result in a bad utilization of the vector registers.

2.6. Message Passing Interface

MPI (Message Passing Interface) is a specification for a standardized message-passing library, which was defined by the MPI Forum, made up by vendors, library writers and application specialists [3].

Message-passing is one of several common parallel programming models. It is typically used for distributed memory but can also be used for shared memory architectures. Each process uses the local memory for computations and exchange data with other processes by sending and receiving messages.

As MPI is only a specification, there are many different implementations with varying performance. Popular ones for C++ are OpenMPI, MPICH or IntelMPI.

3. Implementation

All implementations were added to *large systems 1: molecular dynamics* (ls1 mardyn), a fast and scalable Molecular Dynamics simulation framework, written in C++[6] which is ideal for testing new methods.

The goal was to implement the Half Shell and Midpoint Method as described in section 2.4 and section 2.5 in Mardyn and optimize the performance.

3.1. Traversals

Mardyn already provides different methods for traversing cells and processing the cell pair interactions. Those traversals are implemented via the strategy pattern which allows choosing a traversal in the input file or even at runtime. Additional traversal methods are easy to add.

However, none of the existing traversals utilized a force exchange. So the definition of a traversal had to be extended to include if a force exchange is necessary.

3.1.1. Half Shell

As a Full Shell traversal which uses Newton's third law for calculations between inner cells was already present, this traversal was used as a base to implement the Half Shell method. It was implemented by inheriting from Full Shell and removing half of the calculations between boundary and halo cells – all interactions to backward neighbors (see section 2.4). The Half Shell traversal was flagged to require a force exchange instead.

3.1.2. Midpoint

The Midpoint Method was implemented based on the 63 cell pair interactions for each cell as described in 2.5.2. Those cell pairs are first of all stored as 3D offsets relative to the center cell O (Origin), represented as an array of pairs of three tuples. To reduce the possible mistakes when filling this array with 63 entries and to create more readable code, the offset pairs are not created by hand.

For example, to create the pairs with the opposing side as shown in Figure 2.11, the yellow cube was mirrored at O and then moved on the plane normal to the mirroring direction to create each red cube. This is repeated for three different (not opposite) sides, and all 27 pairs of this kind are created and stored.

Creating the 3D offsets is done exactly once when constructing the traversal so essentially no performance is lost compared to listing all pairs in the source code.

On each domain rebuild (at the start or when the decomposition does load balancing) those 3D offsets are then converted to 1D offsets which represent the shift between two cells in a linear ordering – equal to the index of a cell in the array.

In each time step, all non-halo cells are iterated. Both 1D offsets are added to the index of the current cell separately to get the indices of the two cells that should be interacted. Each index is used to get the cell from the array they are stored in.

3.2. Force Exchange

The force exchange is accomplished by sending force data via MPI to the neighboring processes. The implementation extends the existing code for the particle exchange.

For the transmission, a new MPI data type *ForceData* was created analog to the existing one that is used for transmitting halo copies and leaving particles – *ParticleData*. The new type contains the particle id, the position, and all forces. The adaption of the particle exchange for forces required an additional send and receive buffer as the *ForceData* type and *ParticleData* type differ to much. To reuse the methods that handle send, probe, receive, and so on, they got templated and the data type was used to get the correct buffer.

Only the receive method required additional changes as for *ParticleData* the received molecules are added to the halo cells. For *ForceData*, we have to find the matching molecule in the halo or boundary cells and add the received forces. To not rely on iterating through all those cells we include the position in the transmission. This allows us to easily find the cell the particle resides in.

3.3. Shrinking the Import Volume

Mardyn uses a Full Shell import region for exchanging leaving particles. The same region is used to send halo copies and therefore forces. As the separation of those two regions requires serious refactoring within the domain decomposition, it was not yet implemented; only some preparations.

For the Midpoint Method, one could consider reducing the import region to $\frac{r_c}{2}$ but this requires that no particle moves faster than that in a single time step. Particles that overshoot the leaving particle region get deleted. Hence, the time step size has to be reduced but the smaller import volume probably does not outweigh the time loss in a real simulation.

All benchmarks were performed with the same large import volume.

4. Performance Benchmarks

To compare the performance of the Midpoint Method to Half Shell and Full Shell, benchmarks were conducted on the LRZ's Linux Cluster CooLMUC2. It consists of two 14 core Haswell processors and 64 GB of ram per node, connected via FDR14 Infiniband [5]. Hyper-Threading was disabled for every run, but AVX2 vectorization was used.

For testing, the domain $b_x \times b_y \times b_z$ is filled with a cuboid of particles. The density ρ of this cuboid is $2\,\text{Å}^{-3}$ for *extra dense* input, $1\,\text{Å}^{-3}$ for *dense* input, $0.4\,\text{Å}^{-3}$ for *medium* input and $0.03\,\text{Å}^{-3}$ for *sparse* input. The parameters of the Lennard-Jones potential are fixed to $\sigma = \varepsilon = 1$ and all particles have the same mass.

The number of particles in each simulation results in $(b_x \cdot b_y \cdot b_z) \cdot \rho$ and the number of particles per cell in ρr_c^3 for Full Shell and Half Shell, or $\rho \frac{r_c^3}{8}$ for MP.

We do strong scaling for 1 to 512 processes by doubling the process count each step.

4.1. Cubes of Different Densities

The first benchmark shows the influence of different densities on the performance.

The domain is a cube of $160\,\text{Å} \times 160\,\text{Å} \times 160\,\text{Å}$ with a cutoff radius of $10\,\text{Å}$. This results in $16 \times 16 \times 16$ cells with $1000 \cdot \rho$ particles each for FS and HS, or $32 \times 32 \times 32$ cells with $125 \cdot \rho$ particles for MP.

4.1.1. Without Overlapping

The first tests are done without enabling overlapping.

Extra Dense

For the extra dense scenario, we have 2000 particles per cell for FS and HS and 250 for MP. Figure 4.1 shows the performance in molecule steps per second for different process counts and Figure 4.2 the speedup of MP and HS compared to FS.

The Midpoint Method performs very well and is faster than HS and FS for all process counts. For 1 to 16 processes about 1.3 times as fast as FS and for more processes even up to 1.7 as fast. This is the direct consequence of the way smaller particle count per cell as this results in fewer cutoff checks and a better vector register utilization as described in section 2.5.3.

The Half Shell method performs not as well as expected compared to FS. Up to 8 processes, it is about 1.1 times as fast but for more than that no notable difference

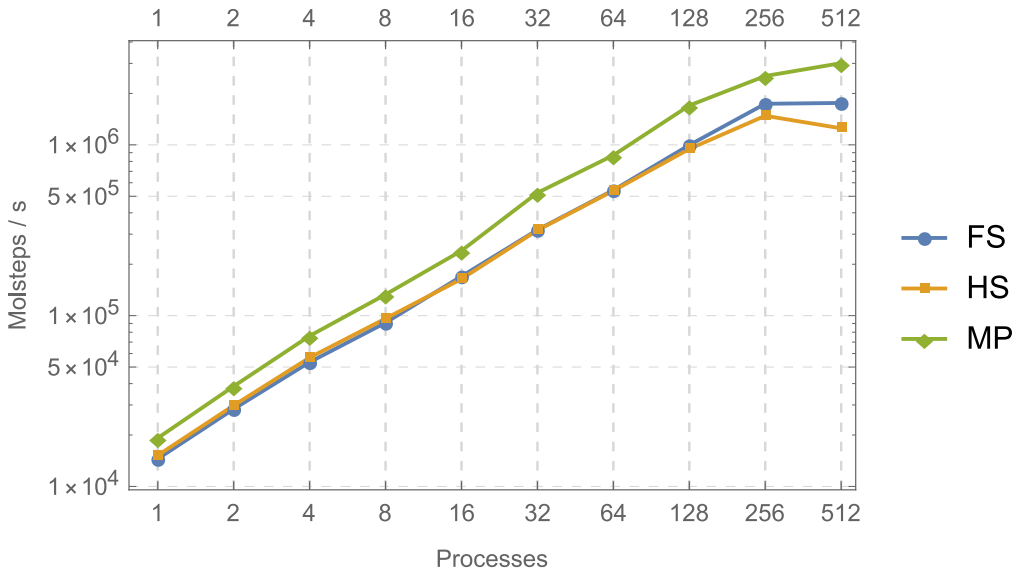


Figure 4.1.: **Extra dense cube**: Performance of the different methods for the extra dense scenario. MP clearly outperforms both HS and FS. No big difference between those two can be seen except for 512 processes.

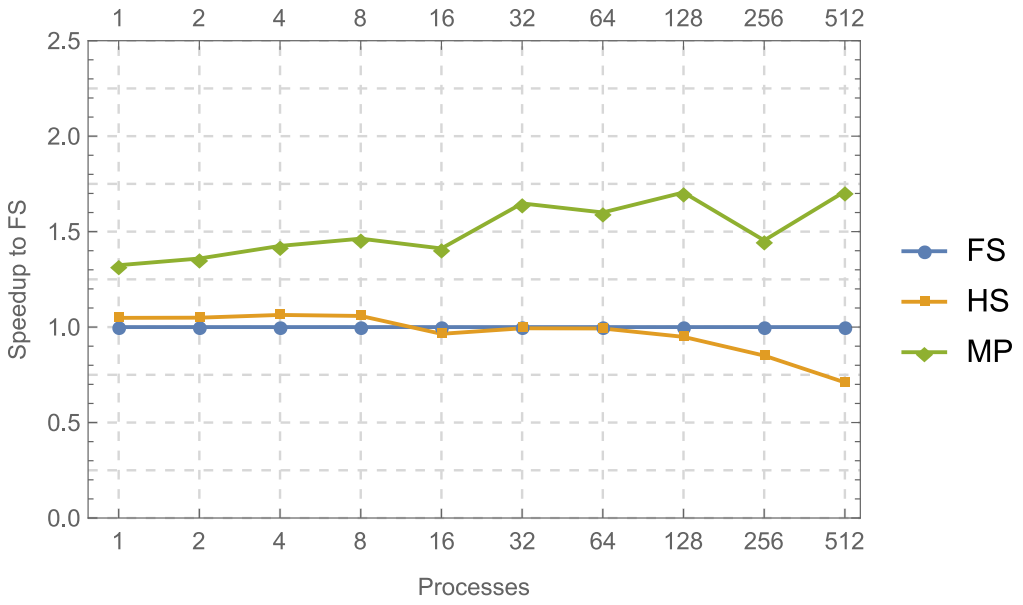


Figure 4.2.: **Extra dense cube**: Speedup of HS and MP compared to the performance of FS. MP is about 1.5 times faster than FS but HS has no advantage in this scenario.

between FS and HS can be seen. For deep parallelization levels, it performs even worse than FS. It seems that the communication overhead is too large for more than 8 processes. The cause for that is probably that the import volume was not halved.

Dense

For the dense scenario, each cell contains 1000 or 125 particles. The results in Figure A.1 and Figure 4.3 are similar to those of the very dense scenario.

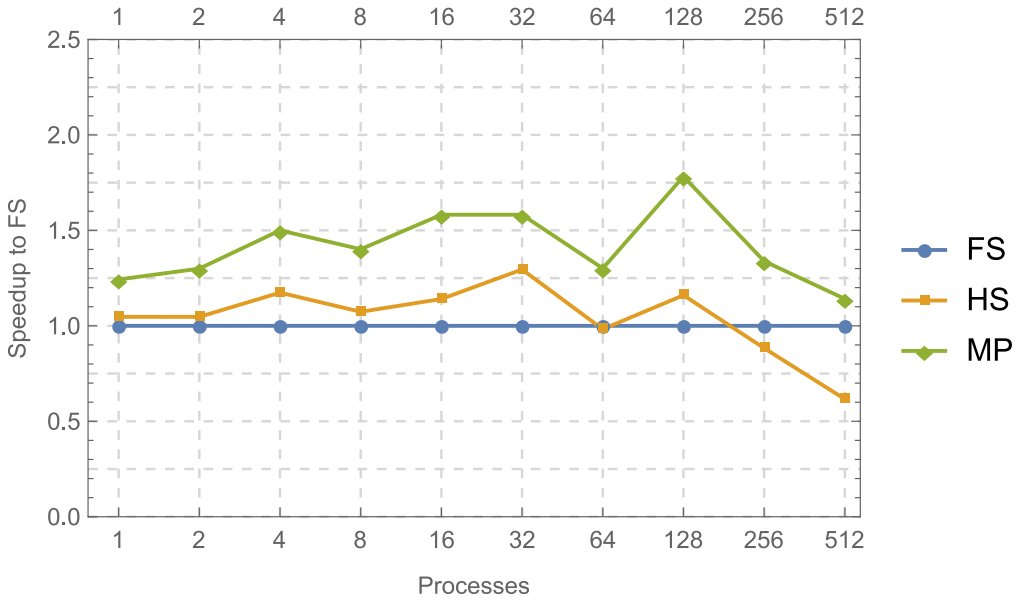


Figure 4.3.: **Dense cube**: Speedup of HS and MP compared to the performance of FS. The advantage of MP is smaller than for the very dense scenario but still quite large. HS performs better than FS for process counts up to 32.

The fastest method is again MP, but this time even HS is faster than FS for most process counts. Up to 128 processes, HS is about 0.1 to 0.4 times faster than FS. Only for 256 and 512 processes, HS drops below the performance of FS and even the advantage of MP shrinks to 0.1.

Midpoint seems to be a good choice for dense scenarios, and Half Shell performs better than Full Shell as well for not extremely dense domains. When reducing the import region, the speed drop for 256 and 512 processes could probably be removed for both methods.

Medium Dense

We further reduce the particle amount per cell to 400 for FS / HS or 50 for MP (see Figure A.2 and Figure 4.4).

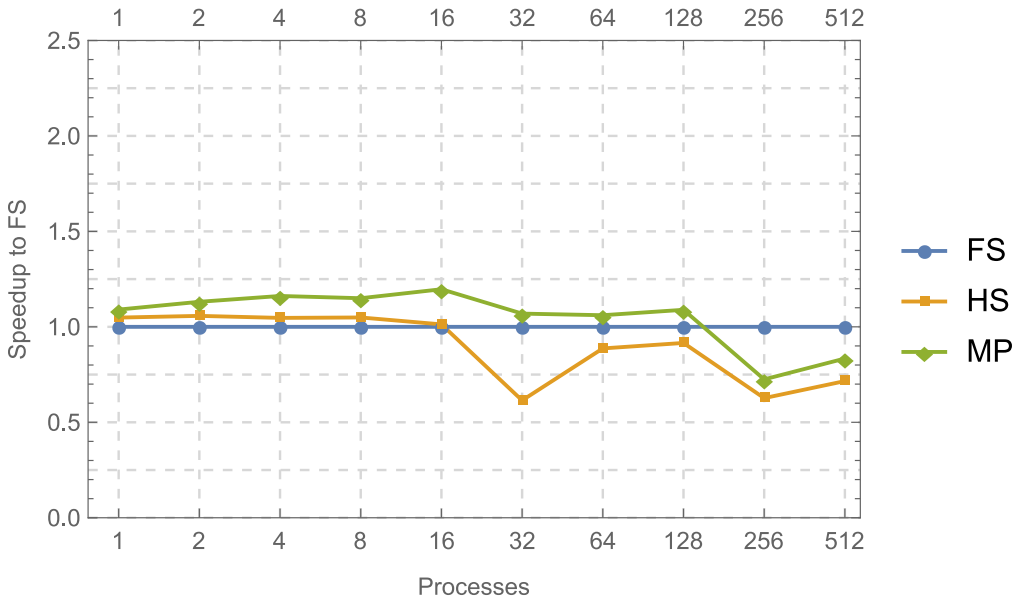


Figure 4.4.: **Medium dense cube**: Speedup of HS and MP compared to the performance of FS. MP has only a tiny advantage, and HS is even worse than FS for more than 16 processes. For 256 and 512 processes, both drop noticeably below FS.

Midpoint still has a small advantage of about 0.5 to 2 over FS up to 128 processes, but after that, it is up to 0.3 times slower than FS. Half Shell drops below FS even earlier – starting at 32 processes.

Somewhere around this particle count, the additional communication takes nearly as much time as the extra computations of the Full Shell method.

Sparse

If we reduce the density even more to our sparse scenario, we are left with 30 particles per cell for FS and HS and only an average of 3.75 particles per cell for MP.

Half Shell performs as the previous results suggest – it starts off slightly better than FS but drops fast and ends a bit worse than FS.

As expected, MP performs way worse than the other methods for so few particles per cell. At this point, no benefit of the method can outweigh the additional iterations necessary for 8 times the cells. The deeper we parallelize, the closer MP gets to the other two though. MP might even catch up for even more processes, but for such low densities, another method is probably advantageous.

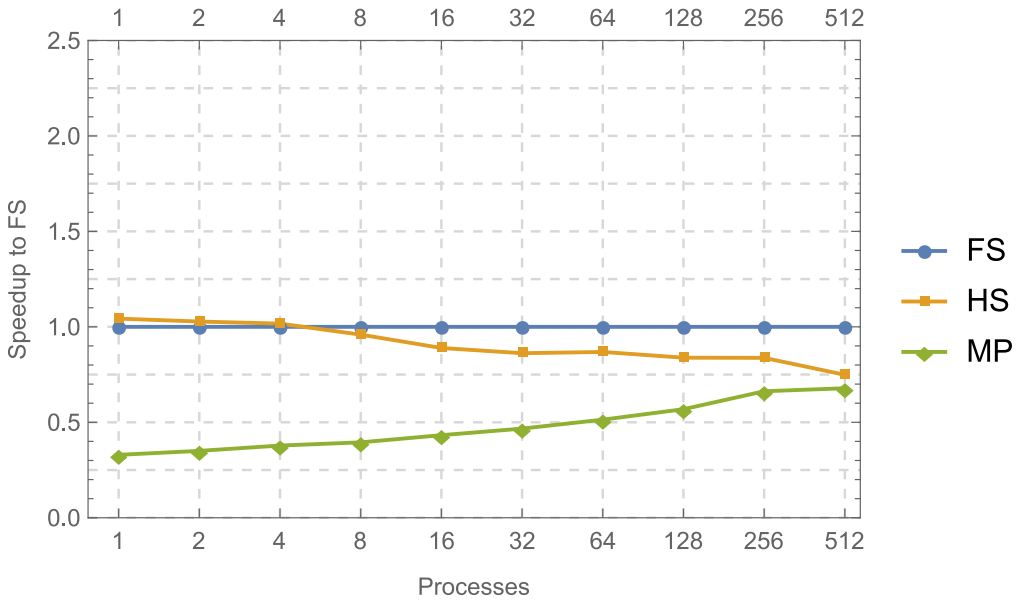


Figure 4.5.: **Sparse cube**: Speedup of HS and MP compared to the performance of FS. MP is 70% slower than FS for a sequential execution but scales quite good to 30% slower for 512 processes. HS keeps getting worse for more processes.

4.1.2. With Overlapping

Midpoint and especially Half Shell perform badly when the amount of communication increases as seen in the last section. The next step was to redo every benchmark from the previous section with overlapping enabled to reduce the time waiting for the communication to finish.

The results can be found in the Figures A.4 to A.10 and Figure 4.6. We want to compare the performance relative to the previous results without overlapping and the speedup between the different methods.

We chose the dense scenario for the comparison.

Speedup

See Figure 4.6 for the speedup with overlapping and 4.3 without.

Enabling overlapping reduces the differences between the methods. FS seems to profit more from overlapping than HS and MP. Especially HS shows essentially no advantage to FS for low process counts, but MP is still about 1.3 times as fast as FS. This even improves for higher process counts.

The most noticeable difference with overlapping enabled is that the drop for the very high process counts does not occur. For 512 processes, MP therefore stays above the performance of FS, at about 1.5 times the speed, and HS only drops to 0.8 times the speed of FS.

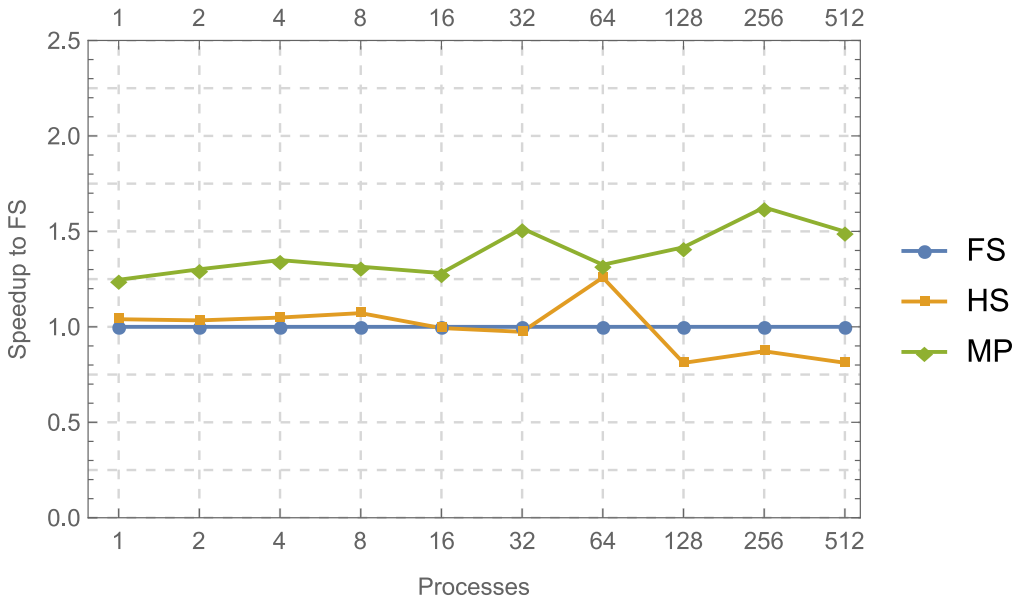


Figure 4.6.: **Dense cube with overlapping**: Speedup of HS and MP compared to the performance of FS. The distance between MP/HS and FS has shrunken, but the performance drop for 256 and 512 processes got removed.

As only the communication for leaving particles and halo copies is overlapped, it seems reasonable that the overhead of the force exchange is still too large for HS to outperform FS.

Relative to Previous Results

We now compare the molecule steps per seconds of the run with overlapping and the run without it. As we can see in Table 4.1, the performance got worse most of the time with overlapping or did not change at all.

There are some process counts, 32 for example in this table, where we see improvements for FS and MP. Half Shell has no significant improvements in the dense scenario. If we calculate these values for the runs of the other densities, we can see similar results except that HS does improve too for some process counts.

It seems that overlapping does not improve the performance in general.

4.1.3. Scaling

One last property we want to observe is how well does each method scale with more processes. We will use the dense scenario for this comparison. Figure 4.7 shows that each of the three methods scales quite well. Half Shell and Midpoint have some problems again for 256 and 512 processes as we already have seen.

processes	1	2	4	8	16	32	64	128	256	512
FS	1.00	1.00	**1.12**	1.02	1.03	**1.25**	0.79	**1.15**	0.67	0.76
HS	0.99	0.99	1.00	1.02	0.90	0.94	1.02	0.81	0.66	1.00
MP	1.00	1.00	1.01	0.96	0.84	**1.20**	0.81	0.92	0.81	1.00

Table 4.1.: Relative performance when enabling overlapping for the dense scenario. Entries with differences of less than 4% are light gray, and the best positive results are bold. Many entries have a value smaller than 1. This shows that the runs with overlapping were often slower than without it. Only in some cases, we have a significant improvement.

Midpoint scales the best in this case for all process counts up to 128 and with a reduced import area probably even further.

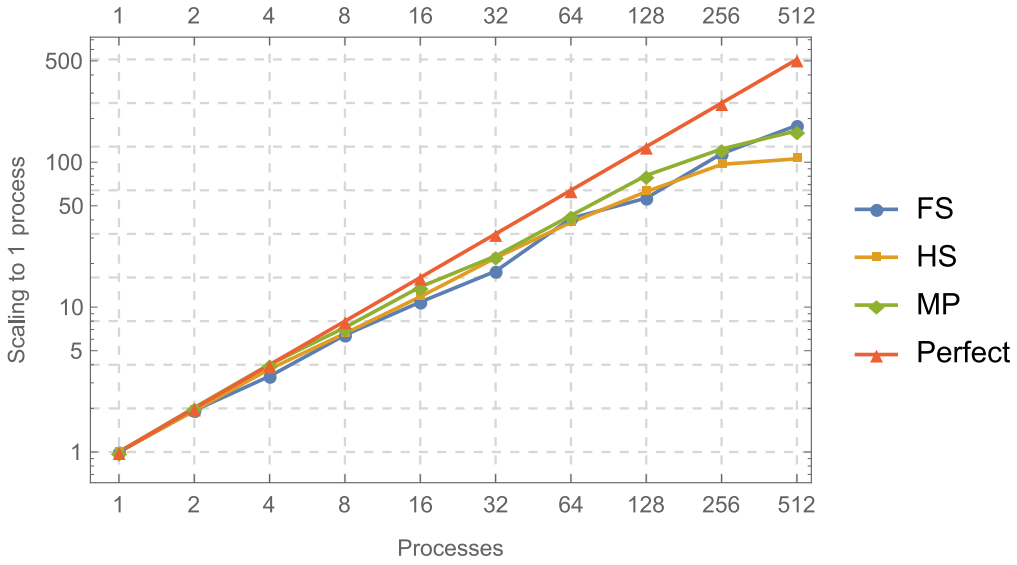


Figure 4.7.: Scaling of the different methods over different process counts. The Midpoint scales best but loses its lead for high process counts.

4.2. Non Cubic Domains

In this section, the influence of the domain shape on the performance is evaluated. The domain has the size of 2560 Å $\times$ 40 Å $\times$ 40 Å with a cutoff radius of 10 Å and a density of 1 Å^{-3}. Hence, we have the same amount of particles in the simulation and per cell as in the dense scenario of section 4.1.

The number of necessary computations stays the same, but the communication is mostly done along the x axis. Figure 4.8 shows the results of this run and Figure A.1 the previous run with the cubic domain.

As we can see, there are only minor differences between both graphs. Midpoint and Half Shell seem to perform a bit worse though. This is probably caused by the communication restriction mentioned above.

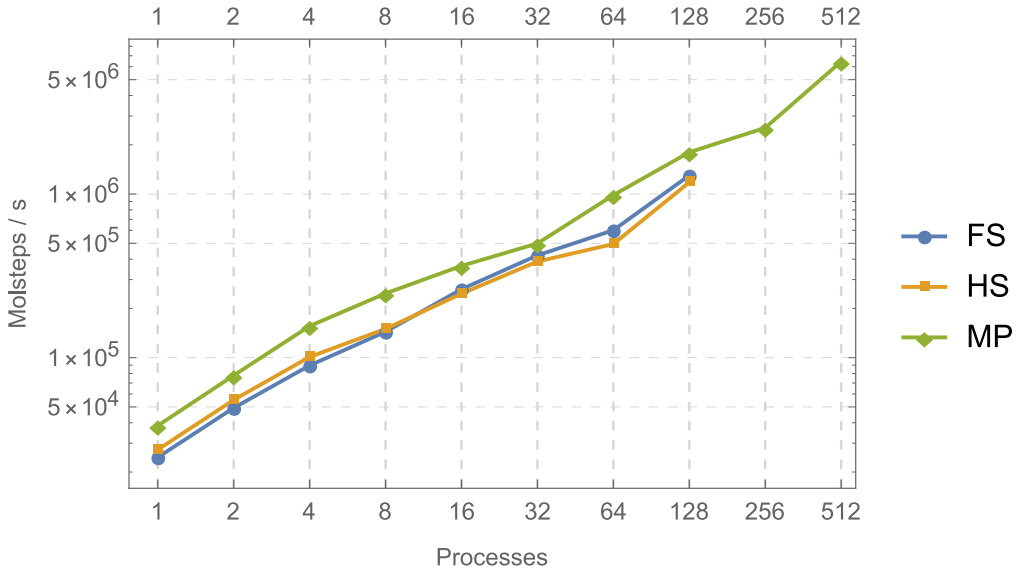


Figure 4.8.: Performance of the different methods for a dense, wide domain. For FS and HS the domain decomposition unexpectedly failed for 256 and 512 processes, but the results for the other values differ not so much from the dense cube.

4.3. Deep Parallelization

As a last test we chose a domain of 40 Å × 40 Å × 40 Å with a cutoff radius of 10 Å and a density of 2 $Å^{-3}$. This results in 64 cells for HS and FS and thus can only be parallelized up to 64 processes. As midpoint has 8 times more cells, we can parallelize it up to 512 processes.

As we can see in Figure 4.9, MP drops a bit below FS for 64 processes. This is the same drop caused by communication overhead we have already seen in section 4.1. For the range of deep parallelization (128 to 512 processes), we then have a larger performance gain. It is unexpectedly large though and it seems as the communication overhead is suddenly gone.

If one would use deep parallelization with lower densities, the Midpoint Method could outperform the other methods when using eight times more processes even if it is slower for the same amount of processes.

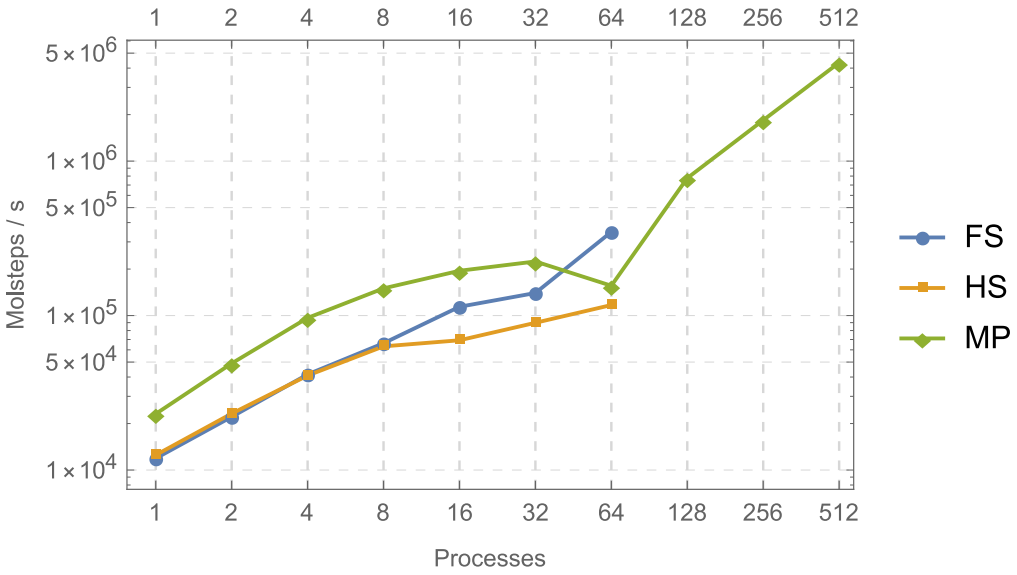


Figure 4.9.: **Deep parallelization**: Speedup of HS and MP compared to the performance of FS. The decomposition of HS and FS fails for 128 and more processes as we can only go as low as one cell per process which is reached for 64 processes. As MP has 8 times more cells, we can parallelize up to 512 processes and gain a better performance than FS with 64 processes.

5. Conclusions and Outlook

The Midpoint Method performs quite well in the tested cases compared to Half Shell and Full Shell across all parallelization levels. Only for very sparse scenarios or small cutoff radii, it dips below the performance of the other two, as the Midpoint Method delivers the best results when we have many particles per cell. The Half Shell method performs not as good as expected compared to Full Shell. The reason for this is probably that the import region was not shrunken and therefore the communication overhead is way too large for deeper parallelization levels. With overlapping the large performance drops for very high process counts can be eliminated, but in general enabling overlapping results in worse performance than without it.

The first step for improving the performance of the Midpoint Method and Half Shell should be to reduce the region for halo copies and force exchange to reach the small volumes described in the sections 2.4 and 2.5. This removes considerable amounts of communication time.

Another possible improvement is to use a hybrid implementation that uses MPI for distributed memory and OpenMP for shared memory. The current implementation only uses MPI and when computations are executed on the same node, the processes still have to communicate with message passing instead of writing into the same memory area.

A. Appendix

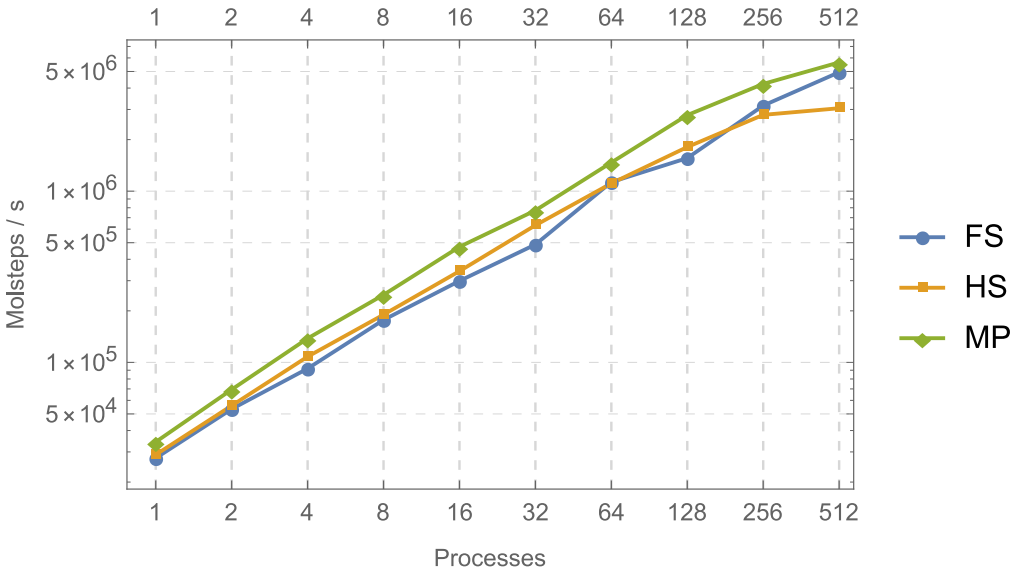


Figure A.1.: **Dense cube**: Performance of the different methods for the dense scenario.

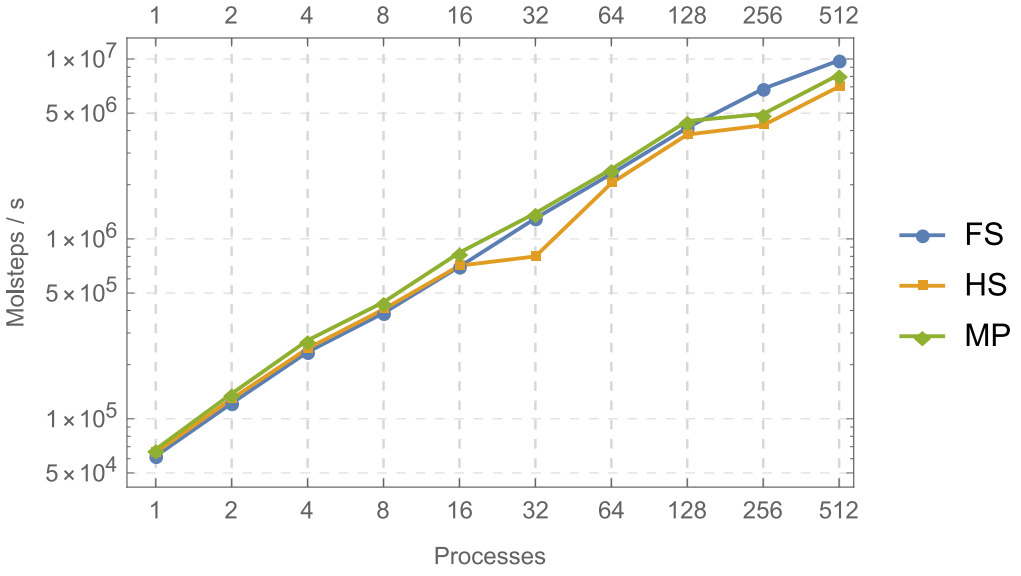


Figure A.2.: **Medium dense cube**: Performance of the different methods for the medium dense scenario.

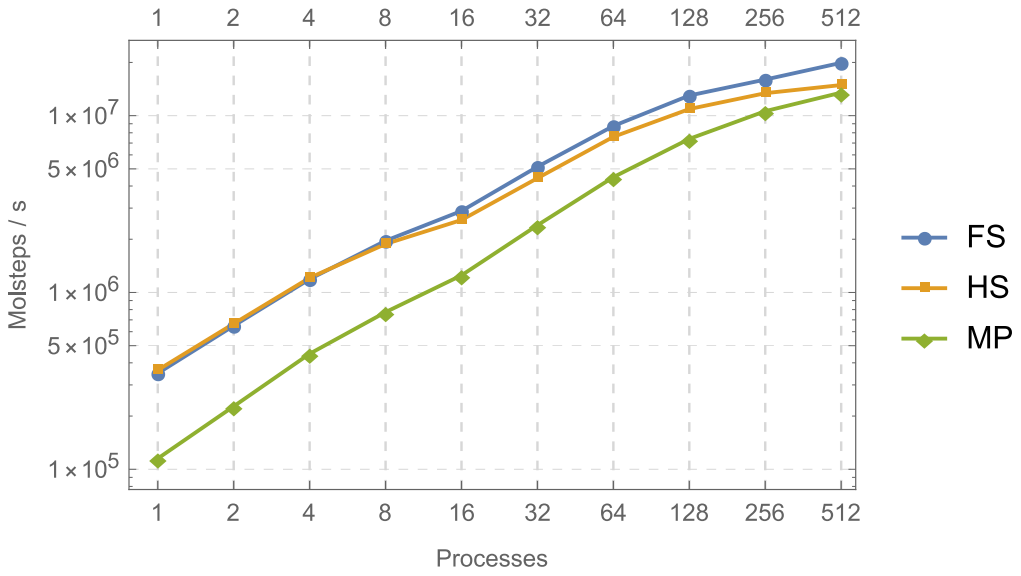


Figure A.3.: **Sparse cube**: Performance of the different methods for the sparse scenario.

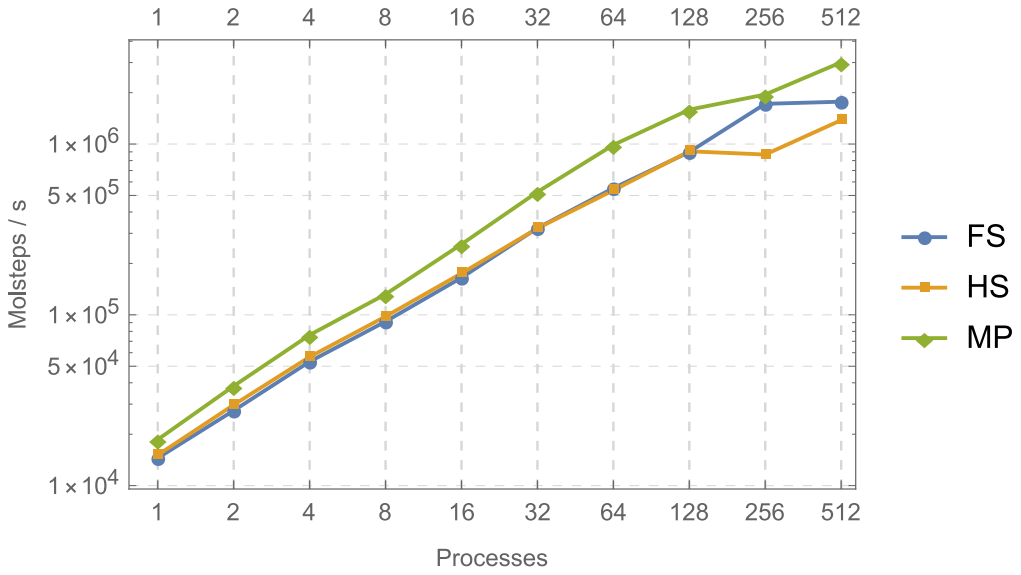


Figure A.4.: **Extra dense cube with overlapping**: Performance of the different methods for the extra dense scenario.

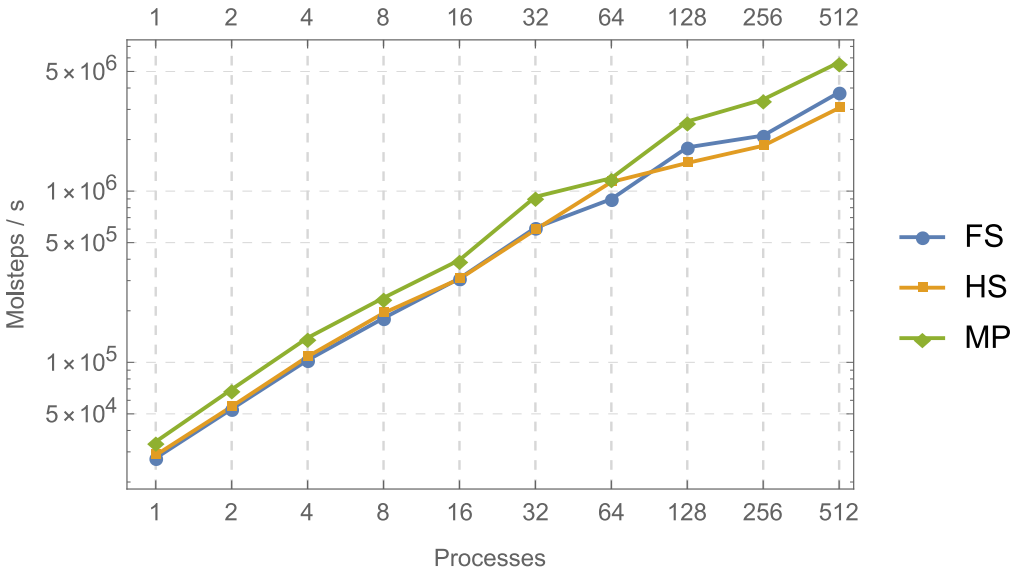


Figure A.5.: **Dense cube with overlapping**: Performance of the different methods for the dense scenario.

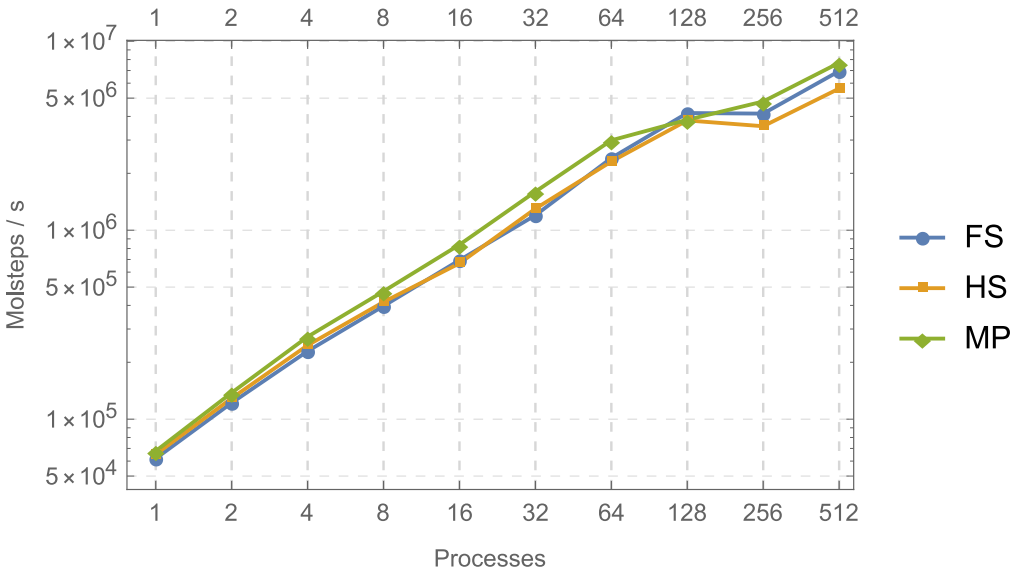


Figure A.6.: **Medium dense cube with overlapping**: Performance of the different methods for the medium dense scenario.

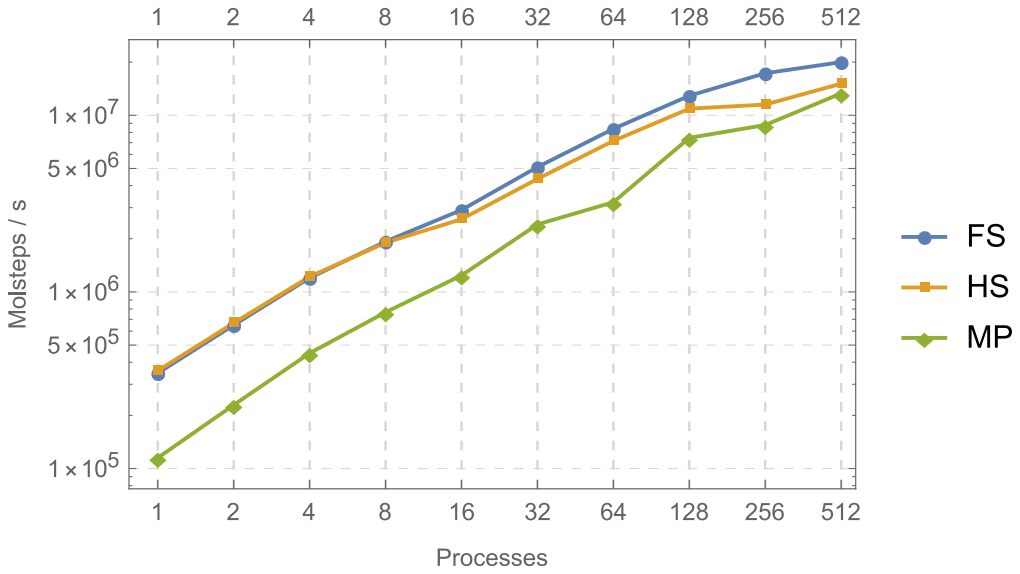


Figure A.7.: **Sparse cube with overlapping**: Performance of the different methods for the sparse scenario.

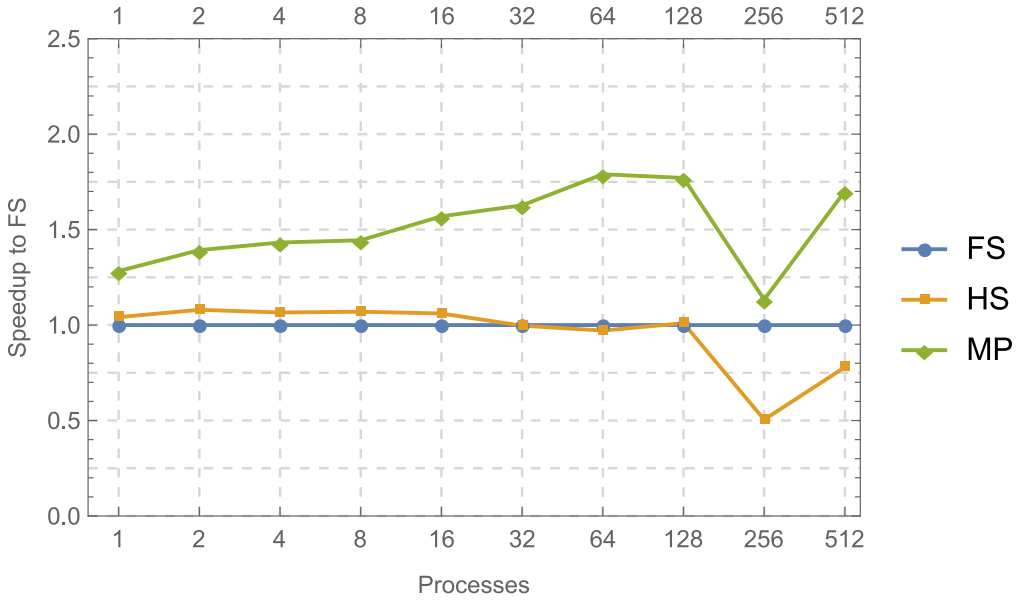


Figure A.8.: **Extra dense cube with overlapping**: Speedup of HS and MP compared to the performance of FS.

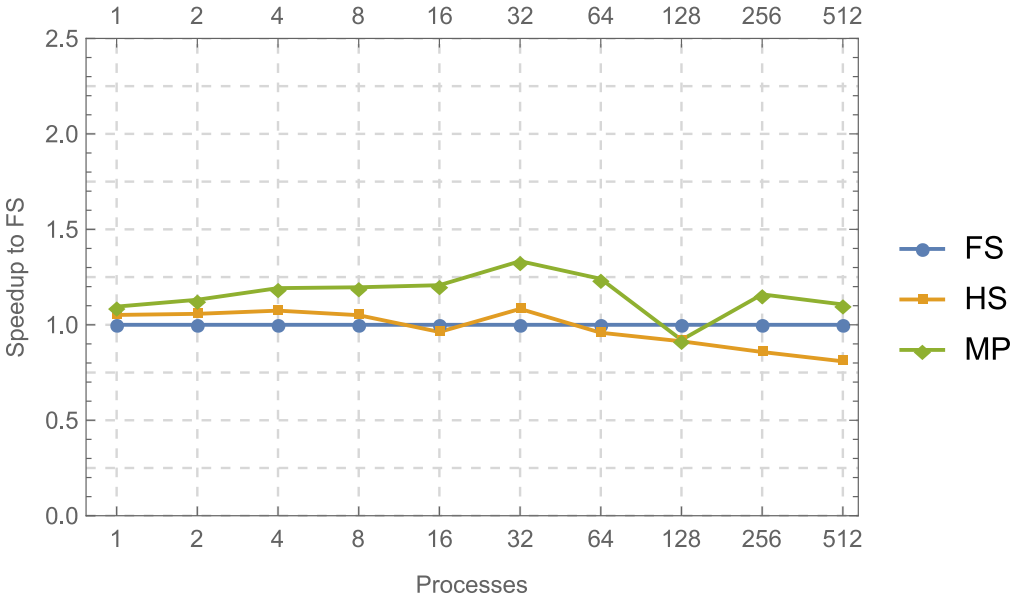


Figure A.9.: **Medium dense cube with overlapping**: Speedup of HS and MP compared to the performance of FS.

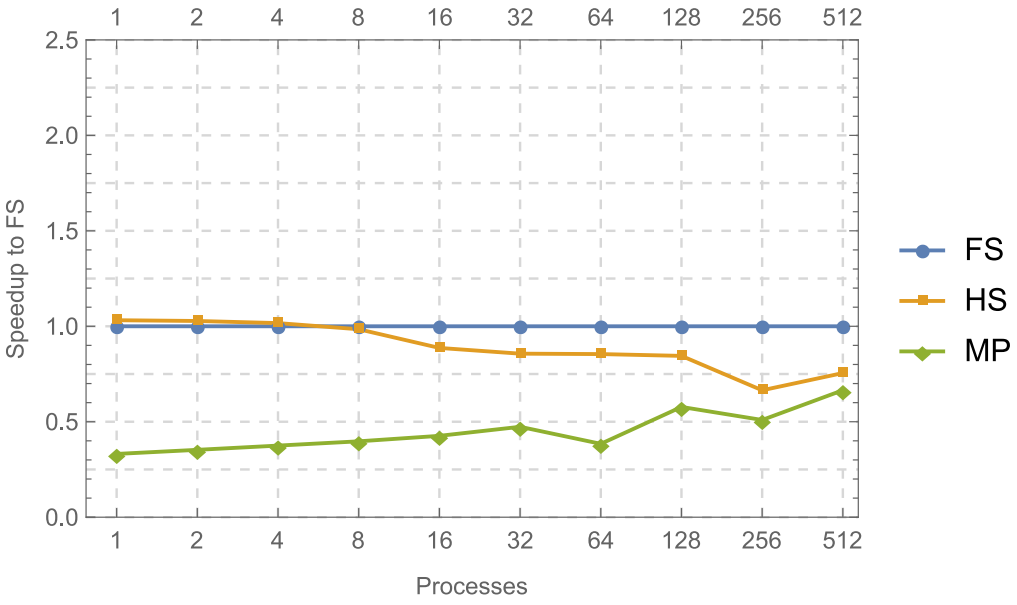


Figure A.10.: **Sparse cube with overlapping**: Speedup of HS and MP compared to the performance of FS.

List of Figures

List of Tables

Bibliography

[1] K. J. Bowers, R. O. Dror, and D. E. Shaw. "The midpoint method for parallelization of particle simulations." In: *The Journal of chemical physics* 124.18 (May 2006), p. 184109.

[2] K. J. Bowers, R. O. Dror, and D. E. Shaw. "Zonal methods for the parallel execution of range-limited N-body simulations." In: *Journal of Computational Physics* 221.1 (2007), pp. 303–329. ISSN: 0021-9991. DOI: http://dx.doi.org/10.1016/j.jcp.2006.06.014.

[3] M. P. I. Forum. *MPI: A Message-Passing Interface Standard Version 3.1*. Ed. by T. University of Tennessee Knoxville. http://www.mpi-forum.org/, June 2015.

[4] M. Griebel, S. Knapek, and G. Zumbusch. *Numerical simulation in molecular dynamics*. Springer Berlin, 2007.

[5] Leibniz-Rechenzentrum. *Overview of the Cluster Configuration*. Aug. 28, 2017. URL: https://www.lrz.de/services/compute/linux-cluster/overview/.

[6] C. Niethammer, S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H.-J. Bungartz, C. W. Glass, H. Hasse, J. Vrabec, and M. Horsch. "ls1 mardyn: The Massively Parallel Molecular Dynamics Code for Large Systems." In: *Journal of Chemical Theory and Computation* 10.10 (2014). PMID: 26588142, pp. 4455–4464. DOI: 10.1021/ct500169q. eprint: http://dx.doi.org/10.1021/ct500169q.

[7] J. Spahl. "Evaluation of Zonal Methods for Small Molecular Systems." Bachelor's thesis. Institut für Informatik, Technische Universität München, Nov. 2016.