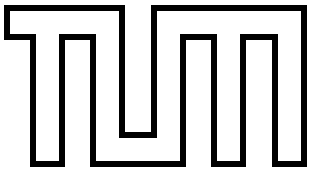


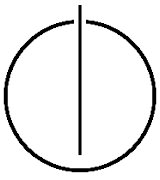
Department of Informatics

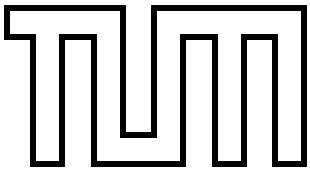
Technical University of Munich

Master's Thesis in Informatics

Parallel Implementation of the Fast Multipole Method

Michael Obersteiner





Department of Informatics

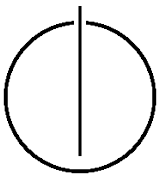
Technical University of Munich

Master's Thesis in Informatics

Parallel Implementation of the Fast Multipole Method

Parallele Implementierung der Fast Multipole Methode

Author:	Michael Obersteiner
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	M.Sc. Nikola Tchipev
Submission date:	October 15, 2016



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

Garching, Michael Obersteiner

Acknowledgments

In this section I want to thank the people that have contributed to this thesis. First, I want to express my gratitude to professor Bungartz for giving me the possibility to write this thesis at his chair and granting me access to the SuperMUC cluster for large scale simulations. Next, I want to thank my advisor Nikola Tchipev for the discussions throughout the semester and the feedback while writing the thesis. Last but not least, I want to mention Martin Obersteiner who gave me helpful suggestions regarding this thesis.

Abstract

In this thesis we develop an MPI parallelization for the Fast Multipole Method in the Molecular Dynamics software MarDyn. Different optimizations to the implementation were investigated to minimize communication overhead. By restructuring of the standard Fast Multipole traversal and the usage of non-blocking communication routines, an overlap between communication and computation is created. Furthermore, synchronized local reduce operations are used to avoid collective operations which significantly improves parallel efficiency for large scale simulations. Moreover, we discuss novel applications of the zonal methods for parallelization of long range interactions in the context of the Fast Multipole Method. Therefore, a new adaptation of the NT method was designed which reduces the communication partners to 6 in the local tree part and to 31 in the global tree part for send as well as receive operations. In addition, import loads are reduced significantly for the global tree part and for up to three levels in the local tree part. In this way, a parallel efficiency of 67% with a speedup of 347 can be obtained even for small simulations with 2 local levels and 512 processors. For larger processor ranges relative speedups of 5.7 for 512 to 4096 processors and 3.6 for 4096 to 32768 processors could be achieved. Moreover, the implementation is compared to the state-of-the-art Fast Multipole library ExaFMM.

Zusammenfassung

In dieser Abschlussarbeit wird eine neue MPI Parallelisierung für die Fast Multipole Methode in der Molekulardynamik Software MarDyn vorgestellt. Für diese Implementierung wurden mehrere Optimierungen entwickelt, um die Kommunikationskosten zu verringern. Durch Veränderungen in der Traversierung der Fast Multipole Methode und durch die Verwendung von nicht blockierender Kommunikation wird eine Überlappung der Kommunikation mit den Berechnungen ermöglicht. Mithilfe von synchronisiertem lokalem Datenaustausch schafft es die Implementierung komplett auf globale Reduktionsoperationen zu verzichten. Dies ermöglicht eine verbesserte Skalierung, vor allem für Simulationen mit großen Prozessorzahlen. Außerdem wurde ein neues Kommunikationsschema entwickelt, um zonale Methoden innerhalb der Fast Multipole Methode verwenden zu können. Hierfür wurden neue Importbereiche für die NT Methode definiert, mit denen die Kommunikationspartner sowohl für das Senden als auch für das Empfangen auf 6 für den lokalen Baum und auf 31 für den globalen Baum reduziert werden können. Dadurch konnte eine parallele Effizienz von 67% mit einem Speedup von 347 für 512 Prozessoren erreicht werden. Für die Testreihen zwischen 512 und 4096 beziehungsweise 4096 und 32768 Prozessoren konnte ein relativer Speedup von 5,7 beziehungsweise 3,6 gemessen werden. Zusätzlich wird die hier erarbeitete Implementierung mit der Fast Multipole Bibliothek ExaFMM verglichen.

Contents

1. Introduction

In our modern society computers have gained enormous influence on scientific research as well as on the personal life. Through the years computers have developed from expensive and large machines to affordable computing units used in smartphones or desktop computers. As the increase of single-core computing power stalled over the past years, new methods have been needed to increase overall performance of computer systems.

A solution to this problem that is studied heavily in the field of High Performance Computing (HPC), is parallel computing. Here scientists try to adapt common algorithms to a parallel computing environment, i.e. many processors that collaborate to solve a problem. Hence, efficient algorithms were needed that try to improve the scalability, i.e. the reduction of computing time with increasing number of processors, as well as the single-core performance through algorithmic improvements and vectorization. Often these two goals are in conflict to each other as the most efficient single-core algorithms might be not suited for a parallel execution. Consequently, researchers have to consider the trade-off between single-core performance and scalability to choose the appropriate algorithm. Furthermore, the problem size often dictates the usage of parallel computation, as single-core computations would not be feasible due to large runtimes.

Numerical simulations are one of the applications that largely benefit from parallel algorithms. Some of the phenomena observed in nature cannot be studied realistically through experiments or experiments might be too costly, time-demanding or dangerous. In these cases numerical simulations are used to predict outcomes of experiments or to test hypotheses. As scientific research strives to simulate more and more realistic scenarios, these computing requirements are constantly growing along with the problem size. Therefore, runtimes are often too large to compute a simulation on a single core.

To satisfy these demands, modern simulations need to efficiently utilize clusters of processors with up to millions of cores. Since these demands will continue to increase, HPC research already targets the exascale where one exaFLOPS, i.e. 10^{18} computations per second, with probably billions of cores will be available for the simulations. Supercomputers already reach around 10^{17} computations per second and these numbers are steadily increasing [4]. The most powerful supercomputer in the world is currently Sunway TaihuLight with a theoretical peak performance of $125,436$ TFlop/s ($= 125.436 \cdot 10^{15}$ computations per second) [4].

Molecular Dynamics [19] is one example of a simulation method that demands enormous amounts of computation. Moreover, it also shows the huge scientific effort to increase problem sizes for more realistic simulations. Recent simulations already reach trillions of particles [15], however, simulations on a large scale, e.g. a cubic meter of gas at

273.15 Kelvin and 101.35 kilopascal which contains $2.7 \cdot 10^{25}$ particles [19], are still out of reach. These numbers demonstrate that there are still orders of magnitude that have to be surpassed to simulate large scenarios at a molecular level. One of the biggest efforts in the field of Molecular Dynamics is therefore to optimize sequential performance of the algorithms through vectorization and algorithmic optimizations. Furthermore, researchers try to design algorithms that are extremely parallel in order to utilize modern and future supercomputers in the best possible way. This is done by optimizing mainly two things: the force calculation between the particles and the communication scheme for the parallel algorithm. The latter also requires an efficient time-integration scheme for moving the particles through space which often requires the particles to be redistributed to other processors.

For the force calculation one has to distinguish between short and long range forces. Short range forces have been successfully computed by introducing a cut-off radius to the force calculation whereas long range forces demand more sophisticated approaches like the fast summation methods. One of these approaches is the Fast Multipole Method (FMM) [18, 21] which achieves an optimal scaling of $O(n)$ where n is the number of particles. Because of these promising properties, the Fast Multipole Method is considered one of the top 10 algorithms in scientific computing [14] right now and it is well suited for exascale computing. In this work, we will try to implement a highly parallel version of the Fast Multipole Method within the MD simulation program MarDyn [11, 23] by optimizing communication costs through efficient data distribution and reduction of communication overhead. This is done by reducing the number of communication partners as well as the communication volume.

1.1. Overview

In chapter 2 we will give a short overview of the theoretical background in Molecular Dynamics. The description will focus on the general algorithm and the computation of the long range forces and especially the Fast Multipole Method. Furthermore, a quick overview of the common parallelization methods in Molecular Dynamics and an introduction to the Message Passing Interface will be given. In addition, the Molecular Dynamics software MarDyn will be presented followed by an overview of the related work. Chapter 3 will then describe in detail our implementation starting from the basic parallel version. Optimizations that include the reduction of communication partners and communication volume through zonal methods [10] and algorithmic optimizations as well as improvements within the MPI framework, will be discussed subsequently. Next, in chapter 4 the results will be presented and an analysis of the scalability will be provided for every optimization step. Finally, the work will be concluded in chapter 5 along with an outlook for future research.

2. Theory

2.1. Molecular Dynamics

The prediction of particle movements is an important part of modern simulations. Historically, the Schrödinger equation was used to described the interactions between particles. However, only in rare cases an analytical solution exists [19]. Therefore, approximations to the Schrödinger equation were proposed, such as the Born-Oppenheimer approximation, which led to less complicated models. Unfortunately, solving the approximate Schrödinger equation is, in general, too complicated for large scale simulations. Consequently Molecular Dynamics use a more drastic approximation by using classical mechanics to simulate the movement of the nuclei of a particle and by calculating the forces through analytical functions which use parameters obtained from experiments or parameter fitting.

The resulting problem, also known as the N-body problem, can be formulated based on Newton's second law of motion:

$$F_i = m_i \cdot a_i \tag{2.1}$$

where F_i is the force that acts on the particle i, m_i the particle's mass and a_i the acceleration of the particle. The result is a system of ordinary differential equations of second order since the acceleration is the second derivative of the particle position. However, this system of differential equations is in general not analytically solvable for more than 2 particles [19]. Hence, numerical methods were invented to solve the N-body systems.

Usually these methods are split into two phases: the force calculation and the time-integration step, i.e. the movement of the particles. The first step is solved by summing up the force contribution of surrounding particles using analytical functions for the particle potentials. This results in an $O(n^2)$ algorithm for pair-potentials as every particle pair needs to be considered. After calculating the forces, a time-integration scheme is used to adjust the particle positions for the next time-step.

The general algorithm for a molecular dynamics simulation can then be summarized as follows [19]:

```
1 while(t < t_end){
2    force calculation
3    //time integration scheme
4    move particles according to forces, position and velocity
5       of particles (maybe with previous values of earlier time-steps)
6 }
```

A popular choice for the time integration is the velocity-Störmer-Verlet scheme [27, 19] which updates the position x and the velocity v at every time-step as follows [19]:

$$x_i^{n+1} = x_i^n + \delta t v_i^n + \frac{F_i^n \cdot \delta t^2}{2m_i} \tag{2.2}$$

$$v_i^{n+1} = v_i^n + \frac{(F_i^n + F_i^{n+1})\delta t}{2m_i} \tag{2.3}$$

Here the superscript denotes the time-step of the respective variable. The benefit of the velocity-Störmer-Verlet method is that it is less prone to rounding errors as other methods, compared to the classical Störmer-Verlet method [19], and that the velocity is available at the same time-steps as the position of the particles which enables a fast calculation of thermodynamic quantities like the kinetic energy [19].

Another important part of an MD implementation is the force calculation which will be discussed in the next section.

2.1.1. Force Calculation

In the general MD algorithm, mentioned in the last section, the forces acting on the particles have to be evaluated. A simple algorithm to achieve this would be a direct summation approach to sum up all pair-wise interactions:

```
1 for all particles p{
2    for all particles p2 != p{
3       calculate forces acting on p caused by p2
4       using analytical functions
5    }
6 }
```

As one can see, the direct summation during the force calculations needs two nested for loops resulting in the previously mentioned $O(n^2)$ complexity of the algorithm. This complexity disqualifies the direct summation for large scale simulation with billions of particles as the amount of computation grows too fast. As numerical calculations, in general, only need limited accuracy, approximations can be applied to reduce the computational work to quasi-optimal $O(n \cdot log(n))$ or even linear $O(n)$ complexity. The applied schemes differ depending on the properties of the specific potentials used for calculating the force.

Short range forces

Forces that decay "fast" in space are called short range forces [19]. An example is the Lennard Jones potential [19]:

$$U(r_{ij}) = \alpha\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^n - \left(\frac{\sigma}{r_{ij}} \right)^m \right] \tag{2.4}$$

where r_{ij} is the distance between particle i and j and $\alpha = \frac{1}{n-m}\left(\frac{n^n}{m^m}^{\frac{1}{n-m}}\right)$. σ and ϵ are constants that are normally determined via parametrization of the specific material of interest. The remaining parameters m and n are usually chosen to be $m = 6$, for simulating the van der Waals force, and $n = 12$ for the Pauli repulsion.

These large exponents are responsible for the fast decay of the potential for large distances which led to the first technique to reduce the complexity of the force calculation. The idea is to simply neglect interactions to particles that are far away. This is done by introducing a cut-off radius r_{cut} [19] which should be large enough to keep the error of the calculation sufficiently small. During the force calculation all the interactions are omitted if the distance is larger than the cut-off radius r_{cut}. As the number of particles in a specific volume is limited, the number of particles in the cut-off radius is a constant. Consequently, only a constant number of force contributions have to be considered per particle which reduces the complexity to $O(n)$.

Finding all particles within the cut-off radius is one of the problems of this method. A naive implementation would check all particle distances resulting in an $O(n^2)$ algorithm. Therefore, suited data structures are required to preserve the linear complexity of the algorithm. Examples are the linked cell method [19] or Verlet's neighbour lists [30]. The latter saves a list of all particles within a certain distance $r_{max} > r_{cut}$ for every particle. As the particles move around in space, these neighbour lists have to be constantly updated. By choosing r_{max} large enough it can be assured that no particle which is not in the neighbour list will by closer than r_{cut} to the particle after a finite predefined number of time-steps. For that reason, the lists are only updated after a predefined number of time-steps. Although having linear complexity, the method requires a lot of book-keeping and updating of the lists.

The linked cell method, on the other hand, applies a domain decomposition by introducing a grid to the domain. An illustration can be seen in fig. 2.1 where one can see the single cells that are formed by the linked cell method. By saving all particles which are inside each cell in a special data structure such as a linked list, one gets an efficient method for accessing particles inside a specific area. If the edge length of a single cell is chosen to be equal to the cut-off radius r_{cut}, it can be guaranteed that all particles that interact with a particle in cell x are located inside of cell x or its direct neighbours. This limits the area that has to be searched to a constant size which again leads to a constant work per particle giving an $O(n)$ algorithm. Similar to the Verlet lists one has to reassign the particles to the cells every time-step as the particles move around in space. However, only a list for every cell and not for every particle has to be updated.

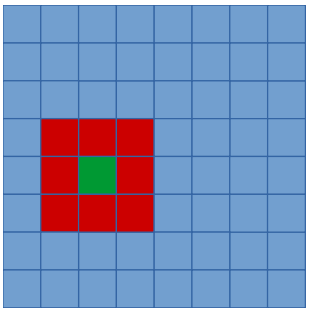


Figure 2.1.: Linked cells structure during the simulation. For the calculation of the forces of particle inside the green cell only the cell itself and the red cells need to be considered if the cell length is at least r_{cut}.

Long range forces

Unfortunately, not all forces decay fast enough in space to allow for a cut-off radius. One example is the Coulomb potential [19]:

$$U(r_{ij}) = \frac{1}{4\pi\epsilon_0}\frac{q_i q_j}{r_{ij}} \tag{2.5}$$

with charges q_i and q_j of the respective particle i and j and the electric constant ϵ_0. The Coulomb potential is used to calculate electrostatic interactions between particles which is often required in Molecular dynamics in addition to the short range forces.

Another example would be the gravitational potential [19]:

$$U(r_{ij}) = -G_{grav}\frac{m_i m_j}{r_{ij}} \tag{2.6}$$

with gravitational constant G_{grav} and masses m_i and m_j of the particles. Thus, fast solver have been studied intensively in the context of molecular dynamics as well as astronomy.

By studying the shape of the curve for the coulombic or gravitational force it can be seen that the potential changes fast for small distances r but changes insignificantly for large distances. Similarly the forces, which are the spatial derivative of the potential, only vary by a small factor for large distances. This leads to the concept of the fast summation techniques that try to summarize the effect of multiple particles in one single force evaluation. Famous fast summation methods are the Ewald summation [29], the Particle-Particle Particle-Mesh (PPPM) [29] and the Fast Multipole method (FMM) [18] with complexities of, respectively, $O(n^{1.5})$, $O(n \cdot log(n))$ and $O(n)$. In the following, we will restrict the description of fast summation methods to the Fast Multipole method as it is the focus of this work.

2.1.2. Fast Multipole Method

In 1985 Rokhlin proposed a method for an $O(n)$ solver for the boundary value problem of the Laplace equation which was a huge improvement over the common $O(n^2)$ solvers [24]. This improvement enabled larger simulations due to the better scaling of the computation compared to the input size. Based on this work Greengard and Rokhlin developed the Fast Multipole Method for the fast calculation of Coulomb or gravitiational interactions that uses similar techniques [18].

The main idea in the algorithm is to divide the contributions of long range interactions into a near and a far field [21]. Interactions to close particles are accounted for in the near field computation whereas interactions to particles that are far away are considered in the far field computation. The key aspect of this separation is that clustered particles (y_1,y_2,..,y_m) interact similarly with a particle x if the distance to the cluster is large. Consequently it is not necessary to calculate all of these interactions separately, but instead they proposed to use a pseudo particle y_0 which interacts with the particle and combines all the effects of the clustered particles. Figures 2.2 and 2.3 show the key difference of the direct summation approach versus the new pseudo particle approach. In order to get the combined effect of all the clustered particles, they introduced the multipole expansion [18, 21] which enables the computation of the Coulombic interactions to a cluster of particles by using a pseudo particle. If the clustered particles have a maximal distance d to the pseudo particle y_0, it is possible to evaluate the force contribution of the cluster at every point that has a larger distance d_{out} to y_0, i.e. $d_{out} > d$ or in other words to all particles outside of the circle in fig. 2.3.

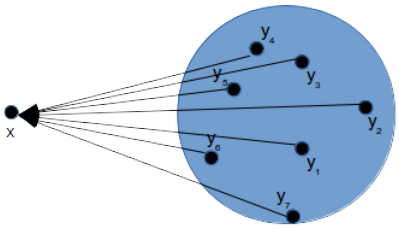


Figure 2.2.: Direct summation algorithm calculating all pairwise interactions of particles [21].

Another observation is that the previous idea can also be applied in the opposite direction. That means that the contributions in the force calculation from particle x to the every particle in the cluster (y_1,y_2,..,y_m) is similar (see fig. 2.4). Again we can use the pseudo particle idea and apply the interactions from outside of the cluster to the pseudo particle and later apply these effects to the particles. This effect from the outside of the cluster is represented by the local expansion [18, 21].

Combining these two ideas we can calculate interactions between distant clusters by calculating these interactions only between the pseudo particles utilizing multipole as well

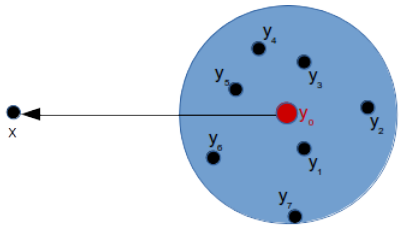


Figure 2.3.: Interaction of multipole expansion y_0 with x that sums up all effects of all particles y_i (M2P operator) [21].

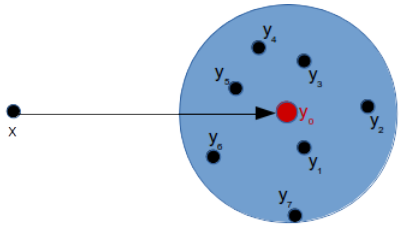


Figure 2.4.: Building local expansion for x at y_0. All y_i may then evaluate the effect of x through local expansion (P2L operator) [21].

as local expansions. However, it has to be ensured that the clusters are "well separated" in order to get guarantees of the error bounds. The term well separated defines an implementation dependant value that represents the minimal distance between two pseudo particles that is required if we want to use the pseudo particles instead of the single particles. This restriction is necessary to get an error bound for the simulation. Greengard and Rokhlin proposed a distance of at least $3 \cdot R$ if every particle inside the two clusters (x_1,x_2,..,x_n and y_1,y_2,..,y_m) have at most a distance R from the respective pseudo particle (x_0 or y_0) which are usually located at the center of the cluster. In fig. 2.5 one can see two well-separated clusters which can interact via their multipole expansions. This reduces the complexity of $O(n \cdot m)$ for the direct summation to the costs for the interactions of the multipole to pseudo particles interactions, which can be done in constant time, the calculation of the multipole expansions and the evaluation of the resulting local expansions at every cluster particle, which costs respectively $O(n)$ or $O(m)$. Hence, the resulting cost is $O(n+m)$ [18].

To get an efficient algorithm we now need to apply this concept in a hierarchical manner. This can be done by increasing the cluster size the further away a cluster is from the particle while still preserving the well-separation criterion. Therefore, the domain [1] is recursively subdivided into eight cubic cells resulting in an octree structure where the children of a node represent the eight equally sized cubes located inside the parent node. This property

[1] we limit ourselves to cubic domains

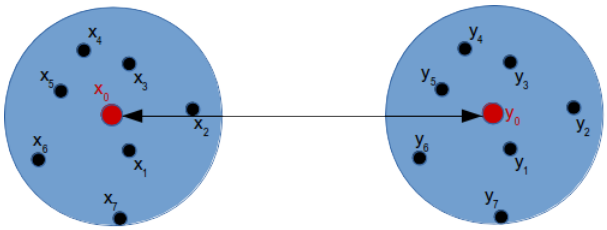


Figure 2.5.: Interaction of 2 clusters via multipole expansions building the respective local expansion (M2L operator) [21].

is illustrated in fig. 2.6 where the tree-structure and the respective subdivided domain is shown for a 2D example using an quadtree instead of an octree. At every center of a cube we now calculate the respective multipole expansion representing every particle located in the cell. This process is repeated for every tree level. Consequently, the higher the level is in the tree the larger the area gets which is assigned to the respective multipole expansion. Using this strategy, well separated can now be redefined as the number of cells ws that must be in between two cells in order to be able to calculate multipole interactions. As a result, no interactions may be calculated to cells within the cube of edge length $2 \cdot ws + 1$ around the center of a cell (see fig. 2.8). Higher values for ws deliver more accurate solutions at lower orders of the multipole and local expansions [21].

A simple algorithm based on this idea [21] would be to calculate the global tree of multipole expansions first and afterwards calculate all the interactions for every particle individually by traversing the tree and summing up the effects of the multipole expansions on the single particle. However, it must be ensured that no interactions are considered twice. Therefore, one approach would be to start at the top level and calculate all the interactions between the particle and the multipoles of subregions which do not contain the particle. Moreover, these subregions need to be well separated to the subregion in which the particle resides. Next, the algorithm goes iteratively down the tree and continues with this idea but discards cells that are children of already processed cells at higher levels. Hence, only children of not well separated cells are considered in the lower level. At the bottom level interactions to the particles within the leaf cells and their neighbours that are not well separated still cannot be computed. This area is called the near field and the direct summation algorithm is applied here despite its $O(n^2)$ scaling. By assuming a sufficiently uniform distribution and a tree size of $O(log(n))$, it can be guaranteed that there is only a constant number of particles in every cell. As a result, there is only constant runtime for the direct summation. To keep the cost of the direct summation to a minimum the distance ws can be set to $ws = 1$ [21]. This decision, however, causes a larger cost for the multipole and local expansions as higher orders are required to guarantee low error bounds.

Unfortunately, this algorithm, which is similar to the Barnes-Hut algorithm [8], does not give the desired linear scaling as traversing the tree of height $O(log(n))$ for every particle

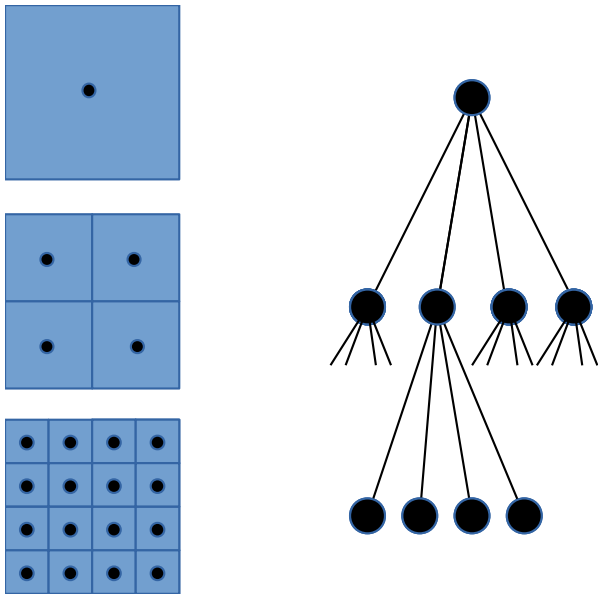


Figure 2.6.: Illustration of the spatial decomposition during the Fast Multipole Method. Left: Domain is split recursively by subdividing it into four equal subregions. Right: Tree representation of the recursive subdivision. Children of a node represent the four subcells located inside of a parent cell.

costs $O(n \cdot log(n))$. In addition, summing up the influences of all particles individually for all multipole expansion on every level would also cost $O(n \cdot log(n))$. To achieve linear complexity five operators need to be used that act solely between multipole and local expansion or act between particles and expansions at the leaf level.

The first one is the Multipole to Multipole (M2M) operator which moves a multipole expansion from one place to another in $O(p^4)$ where p is the order of the expansion. By applying this concept from the bottom to the top of the tree we can get multipole expansions of parent nodes by moving all expansions of the children to the respective parent position and summing them up. As a consequence, only leaf level multipoles have to be initiated by summing up all particle interactions, using the Particle to Multipole (P2M) operator, and every other multipole expansion can be calculated by traversing the tree upwards and calculating the parent expansion from the child expansions. This process is illustrated in fig. 2.7.

The next step utilizes local expansions which is one of the main differences to the previous approach. As in the Barnes-Hut like algorithm we start from the top of the tree and traverse downwards calculating all the interactions between well separated cells. The significant change, however, is that this is not calculated for every particle but instead a local expansion is built for every cell. This can be done by summing up the influence from

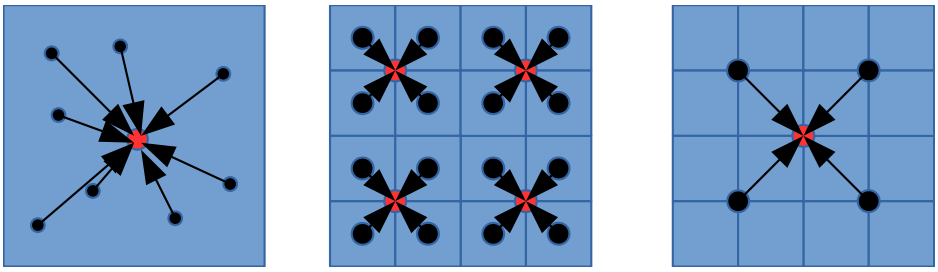


Figure 2.7.: Upwards pass during the FMM traversal. Multipole expansions and particles are marked by dots and red highlighted dots are updated in the respective step. Left: Initial calculation of the leaf level multipole expansions directly from the particles with the P2M operator. Midle and Right: Calculation of higher level multipole expansions from children in the tree using the M2M operator.

the interactions with other cells on the current level. The operator for doing this is called Multipole to local (M2L) operator. Again it has to taken care of that no interactions are considered twice. Hence, only interactions to cells that are children of not well separated cells of the previous level and which are in addition well separated are used. In the resulting scheme for $ws = 1$ a cell interacts with all the cells that are children to a neighbour of the parent of the considered cell or, in other words, the 2 layers of cells that surround the parent area of the current cell (see fig. 2.8). Additionally, the cells of course need to suffice the criterion of being well separated. As a result, there is a maximum of $6^3 - 3^3 = 189$ interactions per cell (for $ws = 1$) on every tree level which are referred to as the interaction list of the cell.

The last phase of the algorithm needs the Local to Local (L2L) and the Local to Particle (L2P) operators. The L2L operator, similarly to the M2M operator, moves the local expansion to another place. Therefore, higher level local expansions can be moved to their children positions and added up with them. By traversing down the tree, the influence of the higher levels can be propagated down to the leaf level. The resulting local expansions at the leaf level now combine the effect of the complete far field for every particle in the cell and by using the L2P operator it is possible to evaluate this effect at every particle position inside of the respective cell. This process can be seen in fig. 2.9 which demonstrates the single steps and also visualizes the similarity to the upward traversal in the first step. The only difference is that now local expansions are propagated down instead of multipole expansions that are propagated upwards.

In addition to those steps, the near field interactions need to be summed up directly as in the Barnes-Hut approach causing constant work per particle which, however, does not conflict with the overall linear complexity.

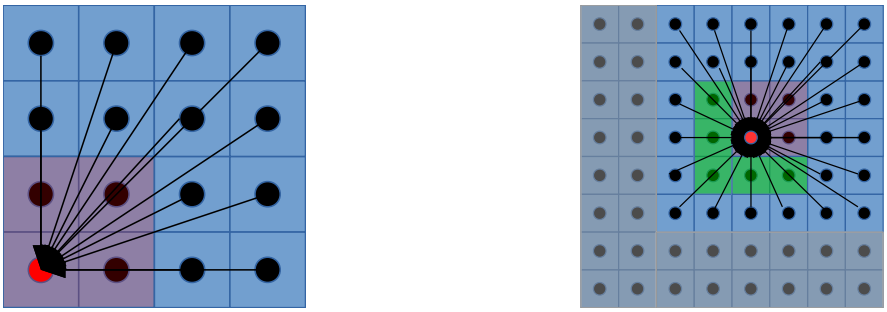


Figure 2.8.: Illustration of the M2L operator during the FMM algorithm. Left: M2L operations that need to be done for the calculation of the red local expansion. The area of the parent cell is highlighted in purple. Right: Larger example that illustrates the concept of well-separated cells. The parent area (purple) and the green area are not well separated for $ws = 1$ and therefore no interactions may be calculated to the included cells. The grey area denotes the area that was already accounted for in multipole interactions on higher tree levels and interactions to this area are therefore ignored.

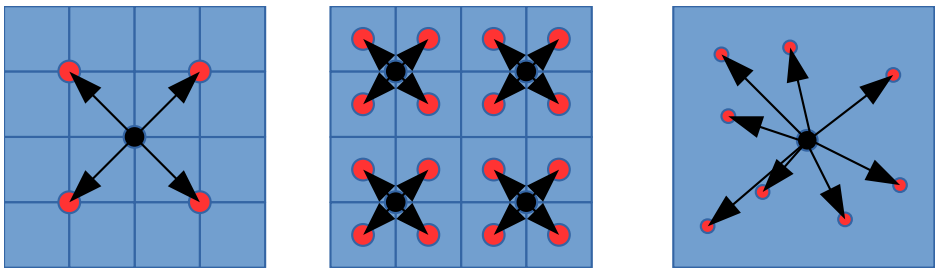


Figure 2.9.: Downwards pass during the FMM traversak. Local expansions and particles are marked by dots and red highlighted dots are updated in the respective step. Left and middle: Local expansions are propagated down starting from the highest tree level with the L2L operator. Right: The combined far field effect is evaluated for every particle with the L2P operator.

Using these operators it is possible to formally define the whole algorithm based on [18, 21]:

```
1 //step 1: P2M
2 for every particle{
3    build initial leaf level multipole expansions
4 }
```

```
5  //step 2: upward pass with M2M
6  for level = level_max - 1:1{
7     //iterate over all cells in curent level
8     for cell = 1:8^level{
9        build multipole expansion of cell from 8
10       children expansions with the M2M operator
11    }
12 }
13 //step 3: horizontal phase on every level with M2L
14 for level = 1:level_max{
15    //iterate over all cells in current level
16    for cell = 1:8^level{
17       for cell2 in interaction list of cell{
18          build local expansion of cell by computing
19          M2L interaction of cell2 to cell
20       }
21    }
22 }
23 //step 4: downward pass with L2L
24 for level = 2:level_max{
25    //iterate over all cells in current level
26    for cell = 1:8^level{
27       add contribution of parent local expansion
28       to cell local expansion
29    }
30 }
31 //step 5: far field computation with L2P
32 for every particle{
33    evaluate far field through local expansion of
34    corresponding leaf cell
35 }
36 //step 6: near field computation with P2P
37 for every particle{
38    for particle in every not well separated cell{
39       evaluate pair-wise potential
40    }
41 }
```

With this formal description we can analyse the complexity of the Fast Multipole algorithm step-by-step. In step 1 we calculate a single P2M operation for every particle which costs $O(p^4)$ where p is the order of the expansion. As the order will be a constant that only depends on the desired error bound, the P2M operation takes constant time. As a result,

step 1 takes $O(n)$ work. The second step now calculates the M2M operation for every cell which also costs $O(p^4)$. Assuming a tree height of $O(log_8(n))$ we have at most $c * n \in O(n)$ cells at the leaf level. The total number of cells can then be calculated by:

$$\sum_{i=0}^{level_{max}} c \cdot n \cdot \left(\frac{1}{8}\right)^i < \sum_{i=0}^{\infty} c \cdot n \cdot \left(\frac{1}{8}\right)^i = \frac{8}{7} \cdot c \cdot n \in O(n)$$

So we get a linear complexity as there is only constant work per cell. In step 3 we have at most 189 M2L ($\in O(p^4)$) operations per cell which is again constant work per cell as 189 is fixed. Similar to step 2, step 4 has one L2L ($\in O(p^4)$) operation per cell which gives again constant work per cell with $O(n)$ cells. The fifth step is analogue to step 1 which gives one L2P ($\in O(p^4)$) evaluation per particle resulting in the same linear complexity. Last, the direct summation step causes only constant work per particle due to the constant number of particles that are not well separated to the leaf cell.

It should be noted that we assume a sufficiently uniform distribution and do not consider any highly clustered scenarios where the number of particles in the surrounding cells might exceed a constant number. For such cases an adaptive method has been proposed [12] to better handle highly non-uniform cases. In this work, however, we only look at the uniform case using the standard Fast Multipole Method.

Another possible pitfall of the algorithm is the high complexity of the operators with $O(p^4)$ which can lead to slow simulations. Therefore, improved versions of the operators have been developed using Wigner rotation matrices reducing the complexity to $O(p^3)$ [32]. There are even approaches that try to reduce the complexity further to $O(p^2)$ ([21] and references therein). An overview of many M2L operator implementations and their complexity can be found in [33]. In this work an FFT acceleration [17] is used in addition to the standard M2L implementation that requires only $O(p^2 \cdot log(p))$ computations.

Dual tree traversal

An alternative way to define the FMM is to use a recursive approach. The resulting algorithm that is a hybrid approach between the standard FMM and a treecode algorithm such as the Barnes-Hut algorithm [8] is called dual tree traversal [13, 34, 33].

In the dual tree traversal there is a distinction between "source" and "target" cells. In our considerations, however, all cells are both "source" and "target" at the same time. At the start of the algorithm one cell that is comparable to the root cell of the FMM is pushed on a stack for source and target cells (see fig. 2.10 on the left). After this initialization, in every iteration a cell pair is pulled from the stack and the larger of the two cells is subdivided. Then for every combination of refined cell pairs, a multipole acceptance criterion (MAC) [13] is applied that decides, based on for example cell lengths and cell distance, if an M2L interaction can be applied or if the cells are too close together. The MAC is similar to the "well defined" criterion of the interaction list based approach mentioned earlier. If the MAC criterion is fulfilled, the interaction is calculated. Otherwise, the cell pair is pushed to

the stack as it needs to be refined. If a cell cannot be refined, a direct summation approach is applied to calculate the interactions between the source and the target. As soon as all refined pairs are processed, the next iteration is started. This process is depicted in the right part of fig. 2.10. The algorithm terminates if the stack is empty. Benefits of the dual tree traversal are the flexibility of the definition of the MAC and the ability to use the method with non-cubic decompositions, which makes it easy to apply adaptivity to the method [34].

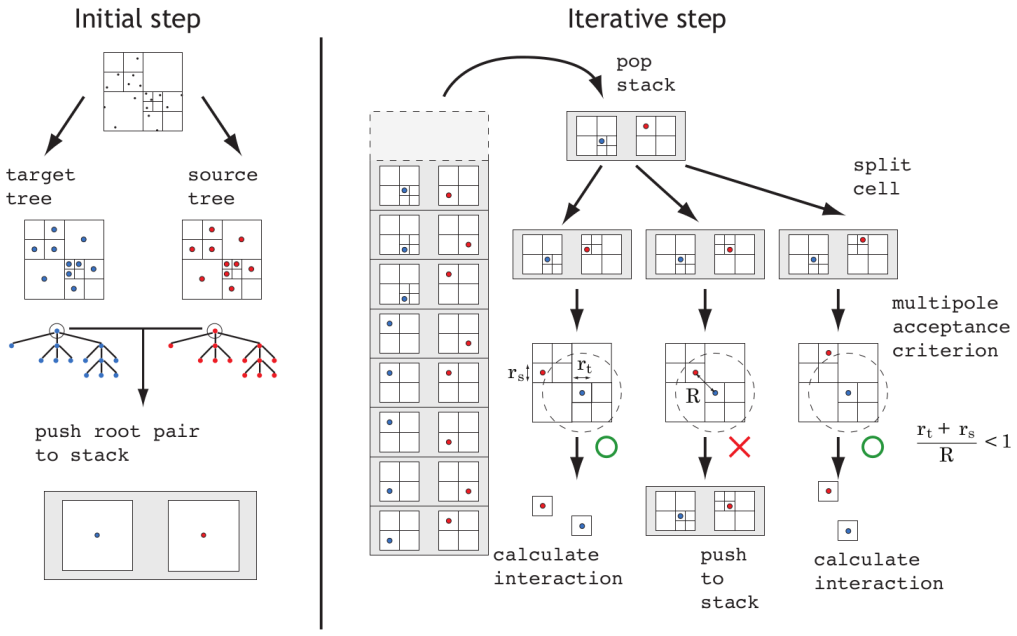


Figure 2.10.: Illustration of the dual tree traversal. Left: Initialization of the source and target tree. Right: General algorithm for processing the stack. The MAC is used to decide if a cell needs to be refined or if the interaction can be calculated. (image source [34])

2.1.3. Parallelization methods in Molecular Dynamics

In the previous sections the basic algorithm for Molecular Dynamics was derived and discussed as it is used in sequential programs. Unfortunately, sequential algorithms do not achieve sufficient sizes and runtimes for many modern Molecular Dynamics simulations. This is mainly due to two reasons: limited computing power of single-core processors which has not increased significantly over the previous years and the limited memory which is available for single-core computations. Both problems can be, to some extend, solved by parallelization. Parallel computers offer large numbers of nodes consisting of

multiple cores which are connected via fast networks, such as infiniband, and large memories which are nowadays often distributed over the nodes [19]. By using parallelization techniques and suited communication routines, algorithms can be implemented which distribute work to the cores of a multicore computer or cluster. Consequently, overall runtimes can be reduced and more memory is available for the complete simulation. Keeping communication costs and additional overhead as low as possible for the parallel strategy is one of the most important parts when designing these algorithms in order to get the best performance.

In molecular dynamics a well-known parallelization strategy is the domain decomposition [19]. Here, the simulation domain, i.e. the virtual area in which the simulation takes place, is partitioned into multiple subareas which are then assigned to a unique processor. There are multiple strategies on how to do this partitioning based on communication costs and load balancing. The standard approach simply generates subdomaines of equal size whereas other methods use more complex subdomains to get better load balancing if particles are not uniformly distributed over space. One example of such an advanced approach is the KD-tree decomposition [9]. In the following we restrict ourselves to the description of the standard decomposition as no other technique is required for uniform distributions which are studied in this work.

Figure 2.11 illustrates the standard domain decomposition for four processors. One problem that arises with this partitioning is that particles which reside at the borders of the subdomains have to interact with particles in neighbouring subdomains during the fore calculation. Therefore, communication is needed as every processor only stores the data of particles that are located in his local subdomain. If this method is combined with the linked cell method (section 2.1.1) and a cell length which is at least as big as the cut-off radius, this means that for short range interactions only the border area, i.e. neighbouring cells which are direct neighbours to the local domain, needs to be imported. Hence, one naive approach is to simply import those cells from all the 26 processors with neighbouring subdomains. Another technique that uses less communications can be implemented in a three step process (see fig. 2.12). First communication is performed in the z dimension communicating the border area to the respective neighbours. Next, the border area and the already received halo cells, i.e. the extended layer of cells outside of the local subdomain, of the x dimension are communicated. This process is repeated in the last step in the y dimension. As a result only six communications are needed but two synchronizations points are created between the single communication steps. The decision between this three-step algorithm or the naive approach has therefore to be made considering the benefit of less communication partners versus the decrease in parallelism due to the additional synchronization points.

Another important aspect of the parallelization is the redistribution of the particles. As particles move through space they might enter a neighbouring domain after a time-step. Therefore, another communication step is needed to transfer particles to the new processor. This can be done by communicating all particles that have moved from the local area to the halo cells to the respective neighbour. A similar approach as in the communication

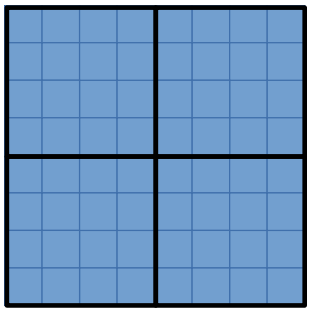


Figure 2.11.: Standard domain decomposition with four processors in a 2D example. The thick lines mark the borders of the local subdomaines whearas the thin lines mark cell borders of the linked cell container.

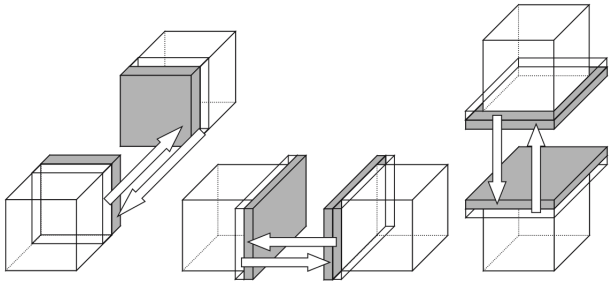


Figure 2.12.: Three-phase algorithm of the border cells. Communication starts in z direction, followed by the x dimension and the y dimension. Already available cells are send in addition to the border cells to reduce the number of communication partners (image source [19]).

step for the force calculation can be used to achieve this by either communicating to the 26 neighbours or by doing a three-phase algorithm using only six communication partners.

This gives a basic algorithm for parallelizing Molecular Dynamics simulations using short range interactions. Parallelization techniques for long range forces will be explained next.

Long range forces

The main difference of long range forces (see section 2.1.1) compared to short range forces is that it is not possible to introduce a cut-off radius, but instead interactions to all particles need to be considered. This is in conflict to the previously introduced domain decomposition which aims to distribute the particles and calculate forces mainly on the local subre-

gions. In addition, a simple halo communication will not suffice for the force calculation as all particles need to be considered.

Fortunately, fast summation techniques such as the Fast Multipole Method have better properties than a naive direct summation algorithm. In the Fast Multipole Method an octree decomposition is used to recursively split the domain into smaller subdomains. This process shows already the similarities to the domain decomposition. A common approach is to assign processors to the nodes on a particular level of the tree. If the number of processors p is a power of 8, i.e. $p = 8^k$, the k-th tree level has exactly p nodes which can be assigned to a unique processor. A concept introduced by [35] now defines the subtree that contains only the children of such a node as the local tree which is completely stored at the respective processor. The tree until the k-th level is called the global tree and the k-th level is referred to as the global level. In fig. 2.13 one can see a 2D illustration of this concept. In addition to the decomposition, data has to be communicated for computations in the global and the local parts of the tree. The communication in the local part of the tree is restricted to the horizontal pass as the upwards and downwards passes solely act on local values. Therefore the two neighbouring layers of multipole expansions need to be imported for the calculation of the local expansions. In the global part of the tree, however, the upwards pass needs communication in addition to the horizontal pass. In the upwards pass the values obtained by the M2M operator need to be communicated and summed up. Different strategies for the horizontal and local part of the tree will be compared in chapter 3.

Zonal methods

Zonal methods [25, 10] aim to reduce import loads and communication partners by allowing updates to particles inside of a halo region, i.e. particles that reside in a neighbouring domain and which are only imported for force calculation, or even to particle pairs where neither of the particles resides in the local subregion. Methods that use the latter criterion are called neutral territory methods. However, these methods should not be mistaken for the NT method [25] which is a specific neutral territory method that introduced the concept of neutral territories. The previous approach, where all cells in the surrounding layer around the local subdomain were imported, can be referred to as the full shell methods. A simple zonal method based on this idea, is to import only half of the cells necessary to finish the force calculation and to allow updates to particle forces inside of the halo area. By communicating these halo values back, it can be guaranteed that all particle interactions are taken into account. This HS method halves the import load and the number of communication partners but requires an additional backwards communication. The HS method is especially suited for few processors as for large processors the import load is mainly influenced by the cut-off radius and cannot be arbitrarily reduced with the number of processors.

An approach that tries to improve the import loads for large numbers of processors is the NT method which was along with the SH method [26] the first neutral territory

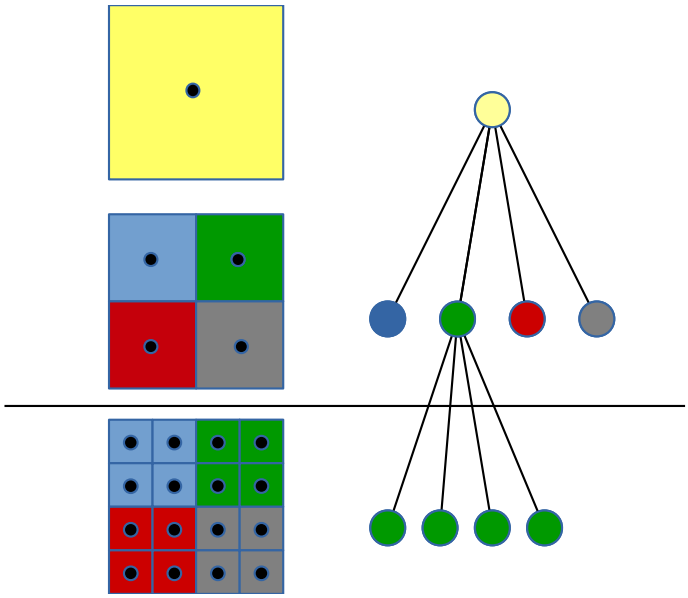


Figure 2.13.: Global (above line) and local part (below line) of the tree for a 2D example with 4 processors. The colours show the respective processor where the tree data is stored. The yellow global part is not stored at a particular processor and needs to be obtained by communication.

method. This discussion will be restricted to the NT method as it has similar properties as the SH method. The NT method utilizes the concept of zones by defining a tower region and a plate region (see fig. 2.14). The tower is the union of the local subregion and the adjacent subregions in the positive and negative y direction that share a face with the local subregion. The plate on the other hand is the union of the local subregion and half of the cells in the xz area of the local subregion that are adjacent to the local subregion including edge neighbours. By interacting all particles inside of the tower with all particles inside of the plate it is guaranteed that all particle interactions are calculated. However, it must be ensured that the particle distances are below the cut-off radius and that the local subregion does not interact with the lower tower because otherwise interactions are calculated twice. As a result the NT method reduces the communication partners from 26 to 6 in addition to a significant reduction in the import load. Moreover, there is only one synchronization at the end of the computation required. One optimization is to omit parts of of the plate that have always larger distance than the cut-off radius to the tower which causes a reduction of the import volume. This process is referred to as rounding [10]. By allowing non cubic subdomains the NT method can further reduce the import load arbitrarily with increasing processors which makes it perfectly suited for exascale computing. It should be noted that in the NT method the communication partners vary

for send and receive as the plate for sending is exactly the other half of the xz area as it is for receiving. Hence, 10 communication partners are required in total of which 6 are used during the send operations and 6 during the receive operations.

The shown examples are part of the two zone approaches as only forces are calculated between two zones. K-zonal methods consider multiple zones and define specific interaction patterns to improve the import loads further (see [10] for more information).

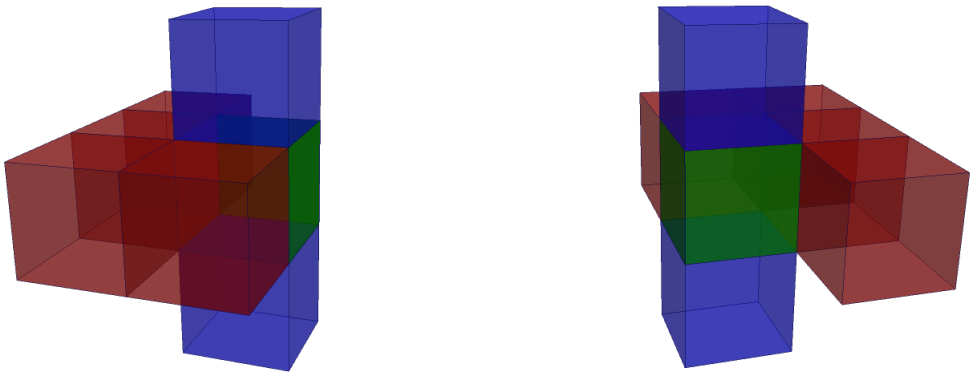


Figure 2.14.: Illustration of the import region for the NT method from different angles [25]. Local subdomain is highlighted green, the tower import region is blue and the plate import region is highlighted in red. Note that both tower and plate contain also the local subdomain.

2.2. Message Passing Interface

The Message Passing Interface (MPI) [16] is a message-passing library interface specification for the programming languages C(++) and Fortran. It focuses mainly on the description of the parallel message passing between processors in shared and distributed memory scenarios but also defines other parallel routines, such as parallel I/O. As MPI is only a formal description of the behaviour of the interface, it does not specify any concrete implementation. Hence, multiple vendors implemented MPI, such as Intel [3], IBM [2] and the open source project OpenMPI [5].

The MPI standard offers multiple ways to communicate between different processors that do not share the same address space. The most common way of communication is the MPI_Send and $MPI\_Receive$ that enable the user to send specified buffers from one MPI process, also called MPI rank, to another. For these send operations multiple versions exist such as synchronous sends that block the communicating rank during the communication or asynchronous communications that return immediately and perform the communication transparently in the background of the program through the MPI library.

By using asynchronous communication, the user, however, has to ensure that necessary communication has finished before using the communicated values but it also allows for an overlap of the communication with the computation which can increase the parallel efficiency. Checking if an asynchronous communication has finished can be done by using wait or test functions (MPI_Wait or MPI_Test). Another way of communication that will be used in this work is the collective operation which enables the user to specify an operation that will be performed on a dataset by a subset of all ranks. Typically these operations are reductions such as summations of values that are distributed over all processors. A common example is the $MPI\_Allreduce$ that performs a reduction and distributes the result to all contributing ranks. The subset of ranks that participate in such a collective operation are organized in communicators. The most general communicator is the MPI_COMM_WORLD that is predefined by the MPI implementation and contains all ranks.

For a more detailed description of MPI, we refer to the MPI standard [16].

2.3. MarDyn

MarDyn [11, 23] is a publicly available Molecular Dynamics simulation software designed for large scale simulations written in C++. It is well suited for the execution on supercomputers even with heterogeneous architectures by using dynamic load balancing schemes. Presently, the calculation of pair potentials for short range interactions such as the Lennard-Jones potential as well as the long range interactions of particles, e.g. the Coulomb potential, is possible. For the calculation of Coulomb interactions the Fast Multipole Method is available with a newly developed FFT acceleration [17] of complexity $O(p^2log(p))$.

As the general Molecular Dynamics step computes short range and long range interactions, the communication of the halo particles, which is needed for the P2P calculation as well as the short range calculation, is already performed before the FMM traversal. Hence, the implementation in this work only communicates expansion values during the Fast Multipole traversal. For this reason, the particle communication is not included in the presented time measurements. However, Eckhardt et al already showed in [15] that the particle communication scales well for large scale simulations with trillions of particles.

2.4. Related work

The high interest in the FMM method caused a lot of research in the past years for optimizing the parallel performance of the FMM algorithm. Ibeid et al [20] uses similar techniques as described in this work using list-based traversal with a separation of a global and a local tree and reducing the number of communication partners to a constant per level. Based on this scheme they evaluated a performance model for the FMM algorithm.

Yokota and Barba [34] use the concept of the Dual Tree Traversal, which was proposed in [13] for the calculation of gravitational forces with treecodes, to increases the flexibil-

ity of the FMM through adjustable MAC values and an easy implementation of adaptive methods for nonuniform scenarios. Auto-tuning features are used to optimize the algorithm based on the specific architecture and GPU parallelization is used to enhance the performance.

In [33] Yokota introduces a task-based threading model to optimize intra-node load balancing and thread-level parallelism for the DTT approach. He showed that his implementation achieved orders of magnitude of increased performance in comparison to other FMM implementations.

Lashuk et al [22] introduced an implementation for non-uniform test-cases on heterogoeneous architecture by using MPI for intra-node communication combined with shared memory and GPU parallelization inside of the nodes. They apply a load balancing scheme using morton-order sorting and redistributing of contiguous chunks to the processors based on the work of Warren and Salmon [31].

3. Implementation

In this section the parallel implementation of the Fast Multipole Method will be discussed in detail showing the applied optimizations and the resulting benefits.

3.1. General concept

The general parallelization technique applied in this paper closely follows the work of [35] using the tree decomposition of the domain with a global and a local tree as described in section 2.1.3. We will now describe the basic implementation which will be then successively improved to get better parallel efficiency.

The current version only supports powers of 2 for the number of processors and distributes the MPI ranks as follows: The global level is computed by $ceil(log_8(p))$ which guarantees at least as many cells on the global level as processors since the number of cells on each level is 8^{level}. For powers of 8, one processor gets assigned to the cell that matches his subdomain in the domain decomposition. For powers of 2 that are not powers of 8 the domain decomposition does not provide cubic subdomains. Therefore, adjacent cells on the global level can be assigned that fit exactly the pattern of the domain decomposition. A 2D example can be seen in fig. 3.1 for powers of 4 and for powers of 2 that are not powers of 4. The levels below the global level contain the local subtrees. Each processor stores the local tree that is rooted at the cell he is assigned to at the global level. If the processor stores multiple cells on the global level, he also stores multiple subtrees. As the concepts are the same for situations where multiple cells are assigned to a processor on the global level compared to those with one unique cell, we restrict the further description to powers of 8 (or 4 in 2D).

In section 2.1.3 it has already been shown that in the local and in the global part of the tree communication is required to finish the FMM computation. The exact communication pattern differs for the global and local part of the tree which will be discussed in the following.

By analysing the FMM algorithm one can see that during the processing of the local tree only cells outside of the local tree are accessed in the horizontal phase while using the M2L operator. Here the two layers of cells that surround the parents cell area are required (see section 2.1.2). As the parent is always located on the same processor, the surrounding cells are either located on the same processor or are directly adjacent to the local subregion. By combining these import regions for all local cells one can easily see that all cells within the two surrounding layers of the whole local subregion need to be imported for every local

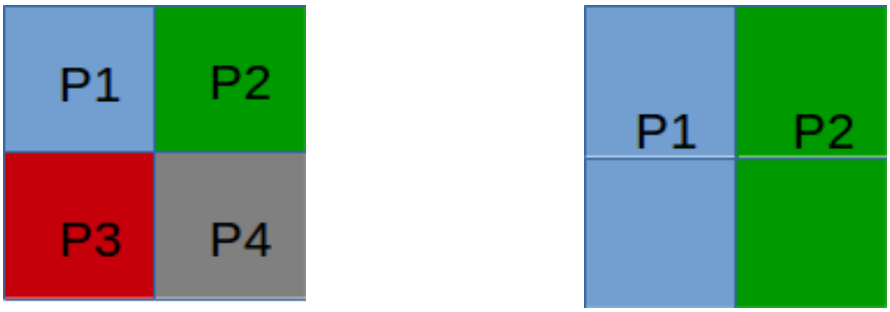


Figure 3.1.: Distribution of the processors on the global level in 2D. Assignment matches area of domain decomposition. The left picture shows the case for powers of 4 the right for powers of 2 that are not powers of 4 where multiple adjacent cells are assigned to each processor.

level. An illustration of this can be seen in fig. 3.2. The communication routine which is used for the import of the cells is the three-phase algorithm described in section 2.1.3. The upwards and downwards pass of the FMM traversal as well as the P2M and L2P operators operate only on parents or children inside the tree or particles inside the local subregion. Hence, no further communication is required in the local tree part.

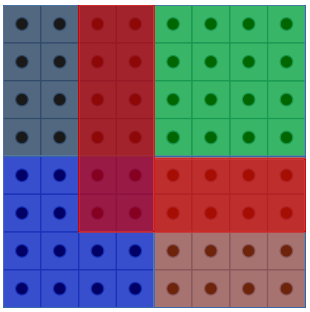


Figure 3.2.: Import region (red) for the M2L operations of the prcoessor with the green subdomain.

In the global part of the tree, the algorithm tries to compute all parent values of the cell starting from the root of the local subtree at the global level. Therefore, communication is required to determine these values in the upwards pass as only one of the multipole expansions is locally available for the M2M operations. This problem is illustrated in fig. 3.3. For the general implementation we use the property that the M2M operator simply sums up all the contributions of all M2M operations during the calculation of the parent multipole expansion. Therefore, one can sum up the values of the 8 processors that calculate the value of the same parent multipole expansion at the level above the global level. Further-

more, it is possible to use the partial values of the parent multipole expansions to calculate a partial value for the higher level. The summation of all these now 64 contributions delivers the correct multipole expansion. This property allows us to calculate first all local contributions of every processor during the upwards pass until the top of the tree. Then a global reduction is performed calculating the complete global tree at every processor by summing up all contributions. As a result, all mulipole values are available for the horizontal phase in the global part of the tree.

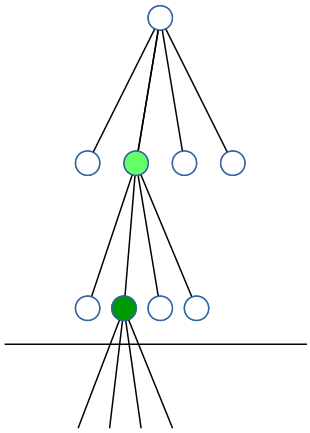


Figure 3.3.: The processor which stores the dark green cell in the global tree can only calculate one of the contributions during the M2M phase for the light green cell. The processors that store the other children of the light green cell have to communicate their contributions to get the final parent multipole expansion.

For the P2P phase the neighbouring cells of the linked cell container are needed as in the short range scenario. Here we utilize the communication which is already performed for the computation of the short range forces described in section 2.1.3. In this, we assume that the size of the smallest FMM cell is $\leq$ one short-range cut-off radius.

The following algorithm gives an overview of all the single steps:

```
1  //Start of upwards pass
2  Step 1:  Calculate local P2M and M2M in local subtree
3  Step 2:  Communicate 2 border layers of local cells to neighbours
4           for every local level (three-phase algorithm)
5  Step 3:  Propagate partial multipole contribution upwards in
6           global tree
7  Step 4:  Allreduce of multipole values in global tree
8
9  //Start of horizontal pass
10 Step 5:  Compute P2P
```

```
11 Step 6:  Compute local M2L (without cells from neighbours)
12 Step 7:  Compute M2L in local subtree to neighbouring cells
13 Step 8:  Compute M2L in global tree
14
15 //Start of downwards pass
16 Step 9:  Compute L2L in global tree
17 Step 10: Compute L2L in local tree
18 Step 11: Compute L2P in local tree
```

3.2. Maximizing Overlap between communication and computation

After analysing the dependencies of the single steps in the basic algorithm one can see that some of the steps are interchangeable. For example step 3 and 4 have no dependency to steps 5, 6 and 7 and therefore the order of those steps can be changed arbitrarily. Furthermore, the order of step 5, 6, 7 and 8 is arbitrarily as there are no dependencies in the horizontal phase between the different M2L operations. However, there are no relative changes in the last four steps possible. Also the last two steps are fixed and cannot be replaced by any other step. Only step 9 of the downwards pass could be computed before steps 6 and 7, but only if step 8 already took place.

With the knowledge of the dependencies we designed an asynchronous algorithm that utilizes the interchangeability of the steps and the asynchronous communication methods of MPI [16]. For maximal performance it is desired that the communication is overlapped by as much computation as possible. Therefore, we designed the algorithm in the way that the communication is started as soon as possible. Hence, the first four steps are kept in exactly the same order as previously mentioned but the communication is replaced by an asynchronous communication scheme. This scheme does not use the three-phase algorithm but a communication with all 26 neighbours (see also section 2.1.3) as the three phase algorithm needs synchronization steps during the communication which complicates the generation of a maximal overlap. In addition the $MPI\_Allreduce$ procedure is replaced by an asynchronous $MPI\_Iallreduce$ which is available in MPI 3 [16].

So far there is only little overlap between the communication and the computation. Therefore, it has to be checked which of the interchangeable steps during the horizontal pass do not require any communicated values. As already noted in the algorithm, step 6 uses only cells in the local subtree and consequently does not need any communicated values. Also step 5 only acts on local particles and particles that have already been communicated before the start of the long range force calculation in the current implementation. Consequently step 5 and 6 need to be processed first in the horizontal pass with arbitrary order. Steps 7 and 8 on the other hand need the communicated values of step 2 and step 4 respectively. Since every of these steps requires only one of these communications to be

finished, they can be started as soon as the respective communication finishes. To improve performance, we implemented a busy waiting routine that checks which of the communications terminates first and then starts the respective step. This can be realized by using the MPI_Test command that returns a boolean value to indicate if the communication has already finished. In this way, waiting times of single processors are minimized.

The order of the downwards pass was not changed even though step 9 could have been placed before the M2L phase in the local tree if the $MPI\_Iallreduce$ would finish first. However, in normal scenarios the $MPI\_Iallreduce$ is by far the most expensive communication step that finishes after the other communications and, in addition, step 9 takes only a small amount of computing time. Therefore, it was not interleaved with the horizontal pass to facilitate the implementation.

It should be noted that the implementation of the algorithm waits until the halo values from all 26 neighbours have arrived. To get a maximal overlap one could even start processing halos that have already been received while still waiting for the other halo values. As the effort for an implementation of this concept would not be in relation to the benefits of this additional overlap, we decided against this concept.

The new algorithm with overlap looks as follows:

```
1  //Start of upwards pass
2  Step 1: Calculate local P2M and M2M in local subtree
3  Step 2: Communicate 2 border layers of local cells to neighbours
4          for every local level (Asynchronous Isend)
5  Step 3: Propagate partial multipole contribution upwards
6          in global tree
7  Step 4: Asynchronous Iallreduce of multipole values in global tree
8
9  //Start of horizontal pass
10 Step 5: Compute P2P
11 Step 6: Compute local M2L (without cells from neighbours)
12 Step 7:
13 while(not both computations processed){
14          if(Isend of step 2 finished){ //using MPI_Test
15             Compute M2L in local subtree to neighbouring cells
16          }
17          if(Iallreduce of step 4 finished){ //using MPI_Test
18             Compute M2L in global tree
19          }
20 }
21 //Start of downwards pass
22 Step 8: Compute L2L in global tree
23 Step 9: Compute L2L in local tree
24 Step 10: Compute L2P in local tree
```

3.3. Removing the collective operations

As mentioned in the previous section, the $MPI\_Allreduce$ as well as the $MPI\_Iallreduce$ dominates the communication cost for large numbers of processors. Hence, a new scheme was implemented to improve the parallel efficiency. Instead of calculating the global $MPI\_Iallreduce$ one can instead use local reduces to get the 8 necessary contributions of the higher level multipole expansions (see fig. 3.4).

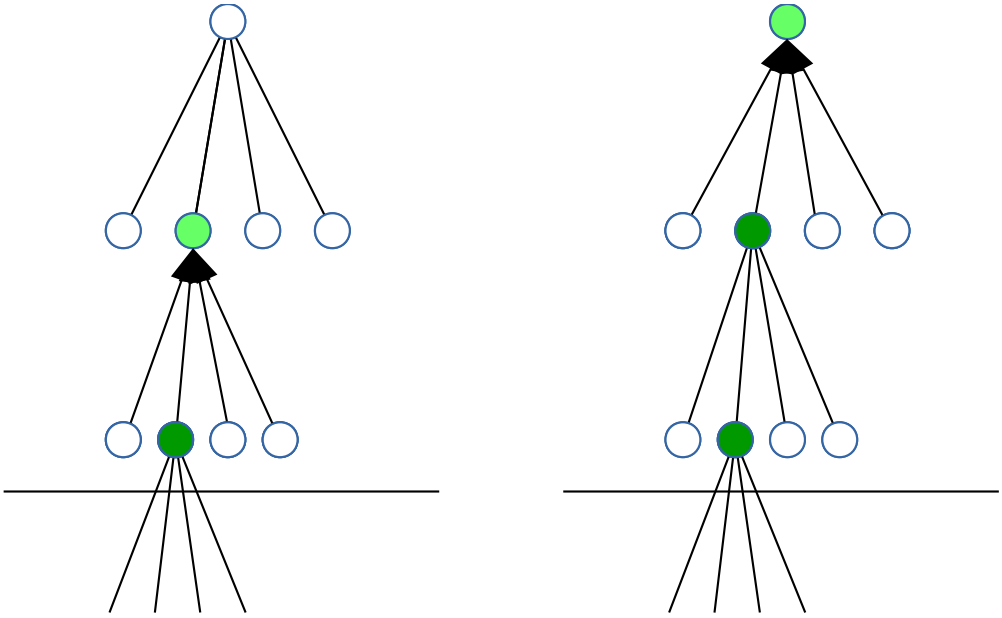


Figure 3.4.: Visualization of the step by step process for avoiding a global $MPI\_Iallreduce$. Starting from the global level a reduce is performed for every group of 4 child processes to finalize the light green multipole value of the next level (left). After a level has been completely finalized the next level is processed in the same manner (right).

Although one can significantly reduce the communication volume with this approach, one now has to wait until values have finalized before calculating the M2M operation for the next higher level. This adds additional synchronization points to the algorithm as otherwise the number of communication partners for the reduce would multiply by 8 for every higher level resulting in a global reduce at the root of the global tree. However, the significant decrease in communication partners to $log_8(p) * 8 \in O(log(p))$ instead of $O(p)$ might be worth the additional synchronizations. Furthermore, the communication volume can be drastically reduced as now only the needed path up the tree is communicated and not the whole global tree.

It should be noted that on every level different processors need to communicate as pro-

cessors that communicated on the lower level now share the same parent node. Therefore, the higher the level the farer the distance to the processors if a cartesian grid of processors is assumed. For simplicity and to use the built in MPI optimization, we have chosen to use $MPI\_Allreduces$ in every step. Since not a global reduction is desired, the $MPI\_Allreduce$ is called with a subset of the processors which only includes the 8 necessary communication partners. Therefore, MPI_Split is used to divide the processors into subgroups which looks as follows:

```
1  int coords[3];
2  int myRank;
3  MPI_Comm_rank(_comm,&myRank);
4  //calculates own position in grid of processors
5  MPI_Cart_coords(_comm, myRank, 3, coords);
6  _neighbourhoodComms = new MPI_Comm[_globalLevel-1];
7  for(int i = _globalLevel-1; i>= 1; i--){
8     int stride = pow(2,_globalLevel - i);
9     MPI_Comm temp;
10    int rowLength = pow(2,i);
11    int colour = ((coords[2] / stride) * rowLength
12       + (coords[1] /stride)) * rowLength
13       + (coords[0] / stride);
14    MPI_Comm_split(_comm, colour, 0, &temp);
15    stride /= 2;
16    colour = ((coords[2] * % stride) * stride
17       + (coords[1] % stride)) * stride
18       + (coords[0] % stride);
19    MPI_Comm_split(temp, colour, 0, &_neighbourhoodComms[i-1]);
20 }
```

Here the MPI ranks are split based on their colour value and saved in the respective communicator for each level in the global tree. In the first split all processors are identified that calculate the same parent multipole value. In the second split operation every processor is grouped with 7 other processors that calculate other contributions to the multipole value. In this scheme every processor communicates with seven processors at every level, even though many processors share the same values at higher tree levels. Furthermore every processor of a group of processors that share the same value communicates with different processors. As it is desired that not single processors communicate more often than others, this approach can be beneficial. However, depending on the specific architecture other approaches could be useful. One approach would be to use one specific processor that communicates for the whole group of processors which share the same value and then distributes this value to all others. This could reduce communications between nodes on clusters. In this implementation the first approach was chosen which will be referred to as local reduce in the following.

If the global reduce is removed from the algorithm, a second communication is necessary to communicate the two border layers of the cell's parent area as now the complete global tree information is not available at the single nodes. Hence, a similar communication scheme as in the local tree part was implemented to solve this issue. However, since every cell now resides on a different processor, we need to communicate with 189 different MPI ranks as 189 neighbouring cells are accessed in the M2L phase. To guarantee that all tree values have finalized, the communication is started at the end of the upwards pass.

The resulting new scheme is described by the following algorithm:

```
1  //Start of upwards pass
2  Step 1:  Calculate local P2M and M2M in local subtree
3  Step 2:  Communicate 2 border layers of local cells to neighbours
4           for every local level (Asynchronous Isend)
5  Step 3:  Propagate partial multipole contribution upwards
6           in global tree:
7           Synchronized reduces with Allreduce in groups of
8           8 processors in global tree
9  Step 4:  Communication of 2 layers of cells in global tree
10          (Asynchronous Isend)
11
12 //Start of horizontal pass
13 Step 5:  Compute P2P
14 Step 6:  Compute local M2L (without cells from neighbours)
15 Step 7:
16 while(not both computations processed){
17          if(Isend of step 2 finished){ //using MPI_Test
18             Compute M2L in local subtree to neighbouring cells
19          }
20          if(Isend of step 5 finished){ //using MPI_Test
21             Compute M2L in global tree
22          }
23 }
24
25 //Start of downwards pass
26 Step 8:  Compute L2L in global tree
27 Step 9:  Compute L2L in local tree
28 Step 10: Compute L2P in local tree
```

3.4. Reducing import loads and communication partners

In this section we introduce different techniques to reduce the number of cells that need to be imported during the FMM and how to reduce the number of communication partners to decrease the communication overhead of the implementation.

3.4.1. Zonal methods

In section 2.1.3 we have introduced the concept of zonal methods to improve import loads and reduce the number of communication partners. As import loads are not proportional to the cell volume but to the number of cells in the Fast Multipole Method, zonal methods cannot be used to get an arbitrarily scaling by reducing the cell volume as for short range potentials. However, the reduction of the import area and the reduction of communication partners can still be beneficial for FMM implementations.

We have shown that in the local part of the tree the two surrounding cell layers need to be imported from 26 neighbouring processors. By using the HS method, we can halve the import volume and communication partners but also introduce a second back communication. Hence, there is no benefit for the FMM as we have the same amount of communications and the same volume in total. The NT method on the other hand reduces the number of send and receive operations to 6 which is an improvement even if the back communication is considered. However, 4 communications are needed along the faces of the subdomain which can increase the overall communication volume as we now need to communicate two times 4 faces instead of only 6. Although this is a problem for large subdomains, i.e. many local levels, the main focus of our optimizations are large numbers of processors with small subdomains. Hence, face communications get smaller in respect to edge and corner computations and a reduction in the communication volume can be obtained. Table 3.1 shows the percentage of saved communication volume at different local levels. One can see that the NT method is beneficial up to the third local tree level - counted starting from the top of the local tree - which is in general sufficient as large numbers of processors reduce the number of local levels. For more than three local level more volume is sent if both communication steps are considered. However the reduction of the number of communications is still preserved for large subtrees.

For the global part of the tree the situation is different as a processor has no longer all cells of the parent area available in his memory. Using a standard NT method with 1 cell is not possible as for different directions the import radius varies depending on where the cell is located in the parent region. Consequently, a new version of the NT algorithm was designed in this work which was adapted to this specific scenario. By considering the whole parent region as the local subdomain of a processor, the standard NT algorithm can be applied by importing the two surrounding layers of cells as in the local part of the tree. The only difference is now that every cell needs to be obtained from a different processor.

Using this approach however would decrease the parallel performance of the implementation as every processor in the parent region would now compute the whole tower

Level	Imported cells	% compared to FS
1	48 (twice)	46.15
2	160 (twice)	71.43
3	576 (twice)	94.74
4	2176 (twice)	111.48

Table 3.1.: Table showing the benefit of the NT compared to the Full Shell (FS) method. The percentage of communication compared to FS includes the backward communication volume.

and plate interaction. Furthermore, the cells of the other processors inside of the parent region would be needed to compute all interactions. Consequently, a modification to the standard tower and plate are needed. As every of the 8 cells in the parent region reside on a different processor, one can split the tower and plate interactions into 8 parts to, on the one hand, reduce the import loads and communication partners and, on the other hand, avoid redundant calculation of M2L interactions. Therefore, the tower is restricted to the subarea of the regular tower that is above and below the local cell inside of the parent region. Furthermore, the plate is reduced to the subarea of the plate that has the same y coordinate as the local cell. Hence, 8 combinations of subtowers and subplates, that compute all the interactions of the NT method, are created. During the back communication, it only needs to be ensured that all the contributions of the 8 different processors need to be added up at the respective nodes. Using this approach the number of communication partners is reduced from 189 to only 24 communication partners for send as well as receive operations. Again the communication partners differ for send and receive as in the standard NT approach. Hence, a total of 40 different communication partners is accessed during send and receive operations. Even though we need to communicate the values back, this is still a huge improvement. Since communication partners are equal to the number of communicated cells for the global tree, the overall improvements are the same for the communication load.

The new algorithm is similar to the last one but includes a back communication after the horizontal pass of the global and the local tree:

```
1 //Start of upwards pass
2 Step 1: Calculate local P2M and M2M in local subtree
3 Step 2: Communicate 2 border layers of local cells to neighbours
4         for every local level
5         (Asynchronous Isend with NT import areas)
6 Step 3: Propagate partial multipole contribution upwards
7         in global tree:
8         Synchronized reduces with Allreduce in groups of
9         8 processors in global tree
```

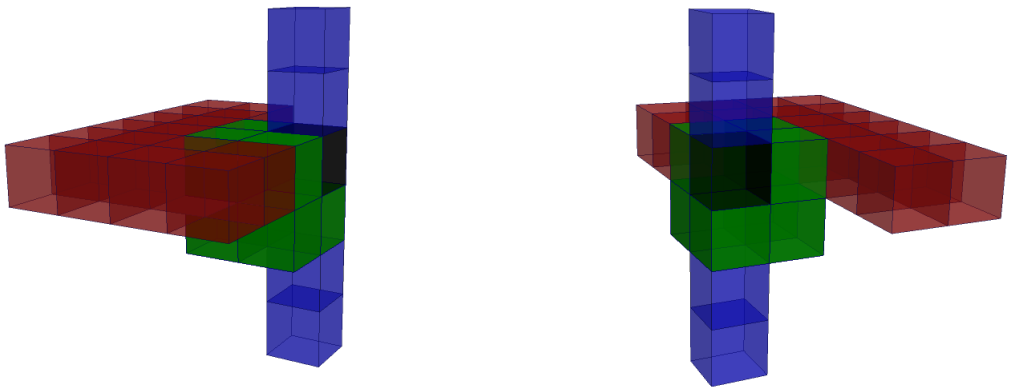


Figure 3.5.: Illustration of the customized NT import zones for the global tree from different angles. The green area is the parent area that contains 8 cells from different processor. The black cell is the local cell stored at the illustrated processor. The red area is the slice of the plate that has the same y coordinate as the black cell. The blue area is the reduced tower that has the same x and z coordinate as the black cell. The green cells which are part of the tower or plate have to be imported too.

```
Step 4: Communication of 2 layers of cells in global tree
        (Asynchronous Isend with NT import areas)
//Start of horizontal pass
Step 5: Compute P2P
Step 6: Compute local M2L (without cells from neighbours)
Step 7:
while(not both computations processed){
        if(Isend of step 2 finished){ //using MPI_Test
            Compute M2L in local tree to neighbouring cells
            Backward communication of halo cells in local tree
        }
        if(Isend of step 5 finished){ //using MPI_Test
            Compute M2L in global tree
            Backward communication of halo cells in global tree
        }
}
//Start of downwards pass
Step 8: Compute L2L in global tree
Step 9: Compute L2L in local tree
Step 10: Compute L2P in local tree
```

3.4.2. Fused communication for global tree

As seen in the previous sections the global part of the tree faces the problem that many cells on lots of different processors need to be imported for the M2L phase. Therefore, normally 189 different cells are used from 189 different processors. There is, however, a possibility to reduce the number of communication partners to 26 as in the local tree part. The method which is based on [20] always aggregates all of the 8 cells that are located in one parent region to reduce the number of communications. Therefore, during the upward pass, when the parent multipole values are reduced for all 8 processors of one parent area, the values of the current level are send in addition to the parent value. As a result, 9 cell values instead of only the parent value are communicated. Hence, every processor now has the information of the whole parent region available at every level without any additional communication.

With these additional cells, we can now reduce the number of communication partners from 189 to 26 by only communicating to one processor for every of the surrounding parent regions. Since there are only 26 of such regions, only 26 sends and receives are needed. To equally distribute the communication work, every processor of a parent region communicates to different neighbouring processors. In fig. 3.6 one can see the main principle of the fuse algorithm for a 2D example.

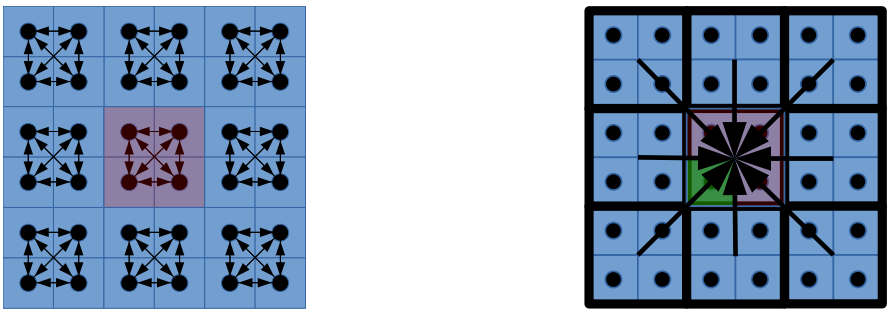


Figure 3.6.: Fused communication scheme for the green cell with purple parent region. Left: processors with same parent region exchange multipole expansions (fuse step). Right: One processor exchanges all four values for every parent region. For every cell in the purple area another processor of the surrounding parent regions sends the fused values.

This method can be combined with the NT method to even further reduce the communication partners to only 6 for send as well as for receive operations. Since the whole parent region is now saved at every processor the standard NT communication scheme can be used without any additional adjustments as mentioned before. However, the computation scheme of the global tree can still be applied to avoid redundant computation. If this NT scheme is used, an additional reduction at the end of the horizontal pass is required to get all the contributions of the 8 processors in a parent region before sending them back.

This is necessary as now every processor inside of a parent region computes different halo contributions and the back communication can only be fused if the aggregated values are known at every processor inside of the parent region.

3.5. Additional optimizations

As described in section 3.3, a method was implemented to avoid the global $MPI\_Iallreduce$ by using smaller reduces and adding synchronization steps inside of the upwards pass. However, for some numbers of processors there might be a benefit of using an $MPI\_Iallreduce$ instead of this approach. To get a more flexible scheme, the algorithm was modified so that the synchronization method can be used up to a certain stopping level and after this level an $MPI\_Iallreduce$ reduces the remaining global tree. Depending on the stopping level this final $MPI\_Iallreduce$ is not a global reduce as already large parts of the tree have been finalized. Therefore, the reduce needs only to consider $pow(8, stoppinglevel)$ many processors. Again by equally dividing all processors with the same values into different MPI communicators, every processor has to communicate to the same amount of other processors. Since we now have a fully adaptive scheme that can be optimized to the concrete scenario and number of processors, an auto tuning step was implemented at the beginning of the simulation to calculate the best stopping level by iterating over one iteration. This optimized stopping level is then used for the whole simulation. As a result, the trade-off between large number of communication partners and delay through synchronization can be handled better.

Another optimization performed in the implementation is to use ready sends instead of standard sends. As in the MD routine normally a synchronization is required every time-step for calculating temperatures or other physical values, we can already start the receiving operations at the end of the previous iteration. Therefore, it is guaranteed that the matching receives have been started before the next FMM calculation is performed. This switch to ready sends improves the delay of the communication which decreases the communication overhead. For backward communications in the NT algorithm, however, standard sends are required as no global synchronizations guarantee that the matching receive operations have already been started.

4. Results

In this chapter a detailed analysis of the results for each implementation step is shown to get a better understanding of the benefits and problems of the different approaches. All tests were performed on a uniform distribution of particles with a constant density throughout the simulation domain with periodic boundary conditions. A standard domain decomposition was performed for all tests, which splits the domain in equally sized cuboid subdomains for each processor. The simulations were performed on the SuperMUC cluster [7] (see table 4.1 for a description of the hardware).

In order to test the implementation for a wider range of parameters, three representative test cases were constructed. The first test case (A) is designed to guarantee two local levels for the maximum number of processors in the respective scaling test. Every smaller number of processors may have more than two local levels. The second test case (B) is designed similarly but the number of local levels is reduced to 1 to get a better analysis for smaller scenarios. Furthermore, this is the smallest possible scenario with only 8 local cells and no local M2L operations. The last test (C) is computed with order 0 of the multipole expansion and two local levels whereas the previous two tests are computed with order 10. Hence, the influence of different sizes and orders can be studied.

As the cluster does not allow for long simulations with small numbers of processors, every scaling is split into three parts: one scaling from 1 to 512 processors with 10 time-steps, one scaling from 512 to 4096 processors with 100 time-steps and, finally, one scaling from 4096 to 32768 processors with 100 time-steps. Furthermore, the different scalings have different simulation sizes to increase the tree height for larger simulations and different number of particles (see table 4.2 for details). If not stated otherwise, all simulations were performed using $O2$ compiler optimizations and SSE vectorization instead of AVX vectorization [1]. However, another benchmark was performed to compare the AVX and SSE vectorization of the final version. The Intel compiler 5.1 was used to build the program and IBM MPI 1.4 to run the code on the cluster. In addition, the newly developed FFT acceleration [17] will be tested to accelerate the M2L calculation. Since the M2L phase is the most time-demanding part of the FMM algorithm, this decreases the parallel efficiency as less computation is available to overlap with the communication.

It should be noted that all runtimes only include the processing of the FMM traversal but exclude all other calculations in the Molecular Dynamics time-step. In particular, the communication of the halo particles is not included as it is already required for the calculation of the short range forces before the FMM step (see section 2.3). All communications

[1] due to technical difficulties

Processor Type	Sandy Bridge-EPXeon E5-26808C
Nominal Frequency [GHz]	2.7
Number of Nodes	9216
Total Number of cores	147456
Total Peak Performance [PFlop/s]	3.2
Total Linpack Performance [PFlop/s]	2.897
Total size of memory [TByte]	288
Total number of Islands	18
Nodes per Island	512
Processors per Node	2
Cores per Processor	8
Memory per Core [GByte]	2 (1.5 avail)
Size of shared Memory per node [GByte]	32
Bandwidth to Memory per node [GByte/s]	102.4
Level 3 Cache Size (shared) [MByte]	2x20
Level 2 Cache Size per core [kByte]	256
Level 1 Cache Size [kByte]	32
Level 3 Latecy [cycles]	30
Level 2 Latecy [cycles]	12
Level 1 Latecy [cycles]	4
Latency Access Memory	160
Interconnect Technology	Infiniband FDR10
Intra-Island Topology	non-blocking Tree
Inter-Island Topology	Pruned Tree 4:1
Bisection bandwidth of Interconnect [TByte/s]	12.5

Table 4.1.: Technical data of the SuperMUC Petascale System [7]

of multipole expansions that are listed in the algorithms of chapter 3 are included in the FMM runtime.

4.1. Initial parallelization with overlap

First, the initial parallel version with maximized overlap described in section 3.2 will be analysed. Figure 4.1 shows the strong scaling results for the different test cases without the FFT acceleration.

One can easily see that the scaling for 2 local levels already scales up to 32 thousand processors for order 10 as well as for order 0. Hence, the order seems not to influence the scaling significantly. This can be explained by the reduced communication volume, as the

Test case	Processor range	Number of particles	tree height	multipole order
A	1 - 512	672799	5	10
B	1 - 512	83974	4	10
C	1 - 512	672799	5	0
A	512 - 4096	5398158	6	10
B	512 - 4096	672799	5	10
C	512 - 4096	5398158	6	0
A	4096 - 32768	43252743	7	10
B	4096 - 32768	5398158	6	10
C	4096 - 32768	43252743	7	0

Table 4.2.: Simulation setup for the three test cases A, B and C with the different ranges of processors.

communication volume scales like $O(p^2)$ where p is the order of the multipole expansion. Consequently, communication volume is decreasing with lower orders as well as the computation time. Apparently, these two effects are balanced well enough to give comparable scaling effects even though the M2L operations scale like $O(p^4)$. The situation is different for simulations with smaller trees. For 1 local level the scaling is reduced significantly and therefore only moderate scaling can be obtained until 4096 processors. The reason for this is the reduction of local computation due to fewer local levels. Hence, communication dominates the overall simulation time for lower processor numbers as less overlap with the computation can be achieved. Furthermore, the portion of the simulation time spent in the global tree traversal increases for less local levels which also negatively influences the scaling as redundant work is done in the global part of the tree.

These observations are in agreement with the waiting times for all test cases (see fig. 4.2). The graphs show the amount of time the processors are waiting in a busy waiting routine for the termination of the communication. In general, larger number of processors and less local levels tend to increase the waiting time. An interesting observation is that even though the order reduction does not influence the scaling as much as the reduction of the tree height, both scenarios have a comparable increase in the waiting time. This supports the hypothesis that besides the waiting time, the processing of the global part of the tree decreases the performance of the FMM algorithm for small local trees. Moreover, for one local level there is no M2L operation within the local tree which also reduces the overlap with the halo communication.

Similar observations can be made if the FFT acceleration is enabled during the simulation (see Appendix A.1). One can see that due to the reduction of computation in the FFT implementation the parallel performance is heavily decreased and scaling can be observed only to a processor number which is around $\frac{1}{8}$ of the number without the FFT acceleration. This matches with the observation that the simulation with FFT acceleration is about seven

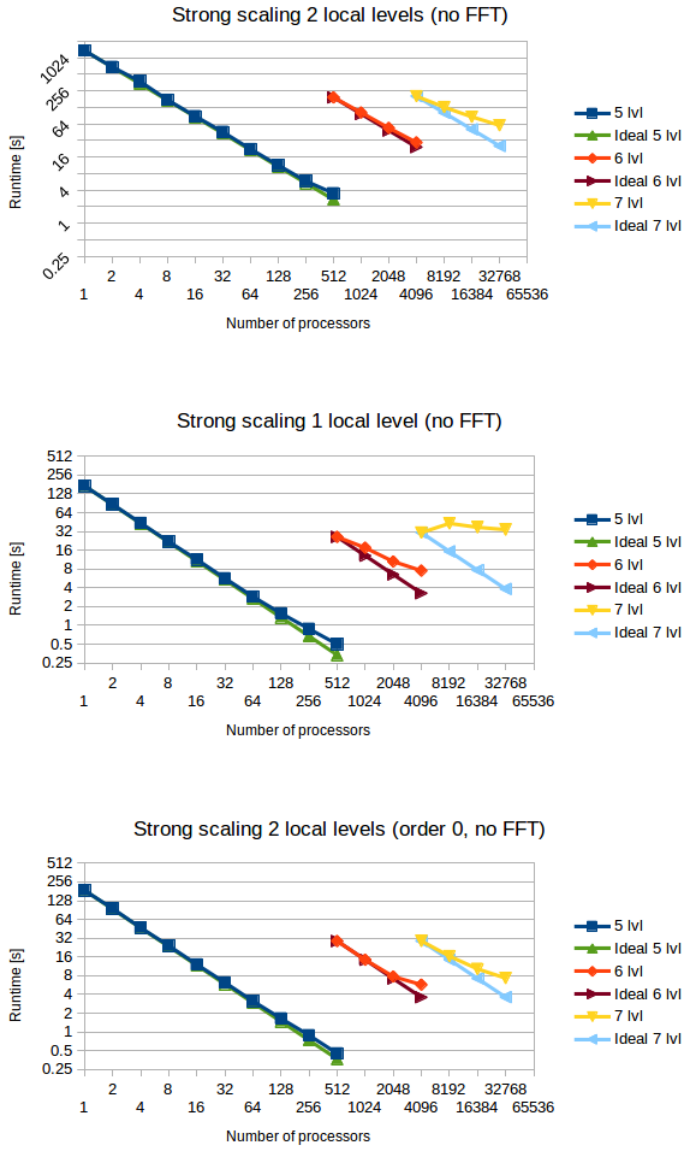


Figure 4.1.: Strong scaling results for test case A (top), B (center) and C (bottom) in the initial parallelization with overlap and without FFT acceleration.

times faster compared to the regular version. For cases where no improvement of the runtime was observed at low numbers of processors, the last range of processors (4096 to 32768) was omitted.

Another difference to the default version is that the FFT acceleration involves pre- and

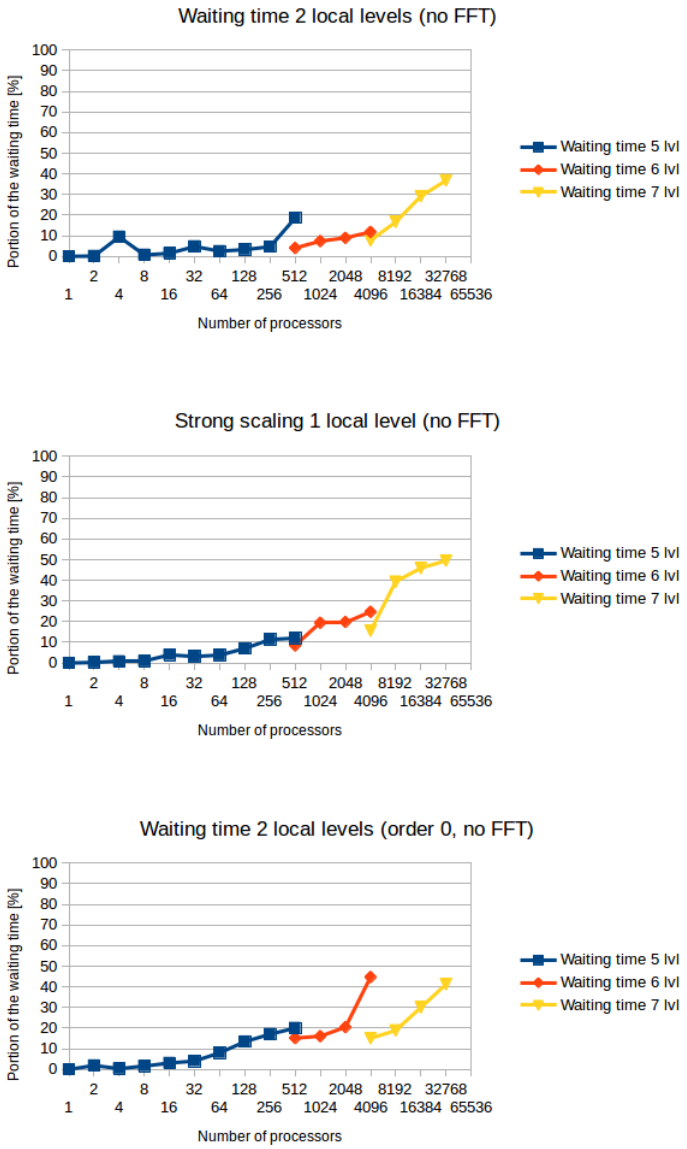


Figure 4.2.: Amount of time waited in test case A (top), B (center) and C (bottom) compared to the complete simulation time of the FMM algorithm.

post-processing steps, in order to convert to and from Fourier space. Hence, the speedup that can be achieved depends on the degree to which those steps can be amortized by the computation. As in the global tree only one M2L operation is computed for every initialized cell, this might lead to smaller speedups in the global tree part. A detailed

analysis of this effect, is, however, beyond the scope of the current work.

4.2. Optimizations

Besides the initial version with overlap, the single optimizations described in chapter 3 were tested on the SuperMUC cluster. Table 4.3 lists all tested optimizations along with their abbreviations. A general overview of the results can be seen in fig. 4.3 for test case A and in fig. 4.4 for test case B. An overview for test case C can be found in appendix A.2. The figures show the speedup of the single optimizations compared to the initial version with overlap for the specific processor numbers, i.e. the speedup is calculated by dividing the runtime of the initial version by the runtime of the respective optimization. These speedups should not be confused with speedups versus respectively 1, 512 or 4096 processes. If no runtimes were measured in the initial version caused by the lack of any scaling for large processor numbers, the speedups are calculated based on the first optimization (locRed) with the removed global reduce operation. It can be seen that for almost every processor number except for 1 processor the optimizations perform better. We now describe the observed benefits of the different optimizations in more detail based on figs. 4.3 and 4.4 and appendix A.2 where the single strong scaling results and waiting times can be found for all test cases and optimization levels.

Abbreviation	Optimization
initial	initial version with overlap
locRed	local reduces instead of global reduce
NT	NT import areas are used
AT	auto tuning of starting level for global reduce
AVX	AVX vectorization is used instead of SSE

Table 4.3.: Abbreviations of different optimizations.

The first optimization (locRed) is the change from the global $MPI\_Iallreduce$ to local reduces with 8 communication partners described in section 3.3. In appendix A.2 the strong scaling results and waiting times are shown for the different test cases. These results show that the optimization already improves the scaling and decreases the waiting times significantly (especially for 2 local levels). This benefit is maximal for high numbers of processors as the global reduce dominates the runtime in these cases. As a result, scaling of the runtime can be observed to much higher processor numbers, i.e. 32 thousand for 2 local levels and moderate scaling until 4096 for 1 local level. The lack of a collective operation as well as the reduction of communication volume, therefore, outweighs the decrease of the parallel efficiency due to the additional synchronization points at every global level. However, there is no improvement in the number of communications which is approximately $O(log(p))$ for the $MPI\_Iallreduce$ and for the new scheme. Hence, latency

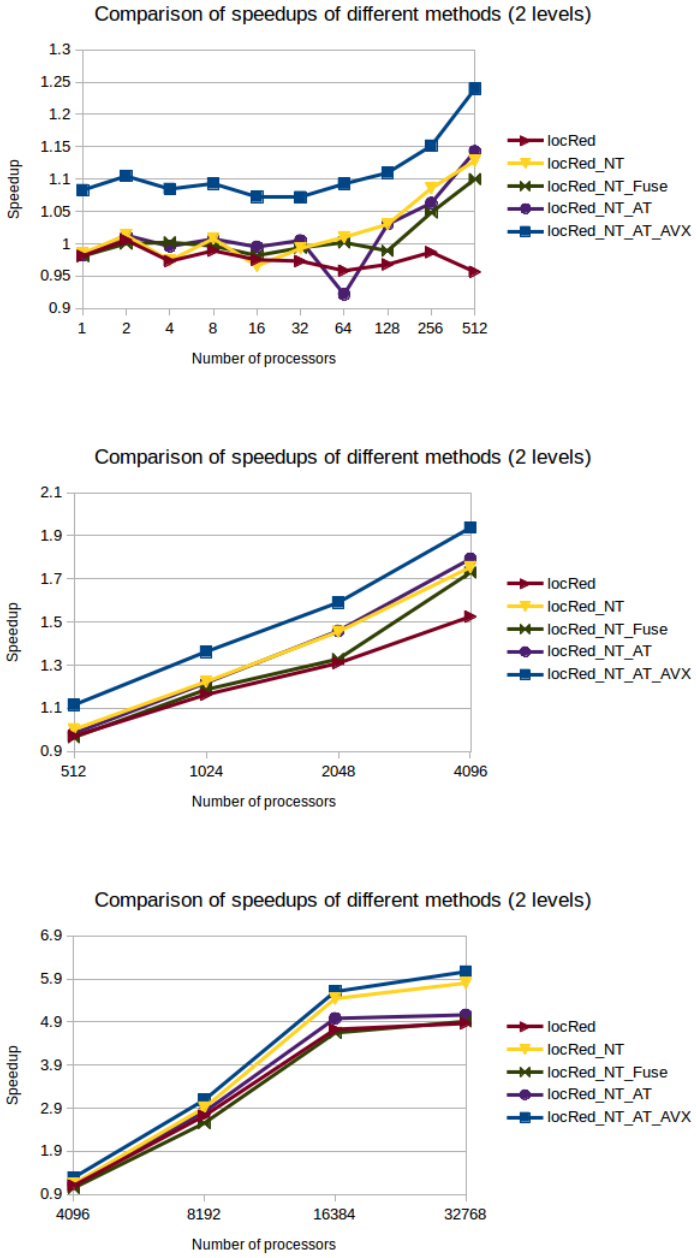


Figure 4.3.: Speedup of the runtime for the different optimization steps compared to the intial version with overlap in test case A.

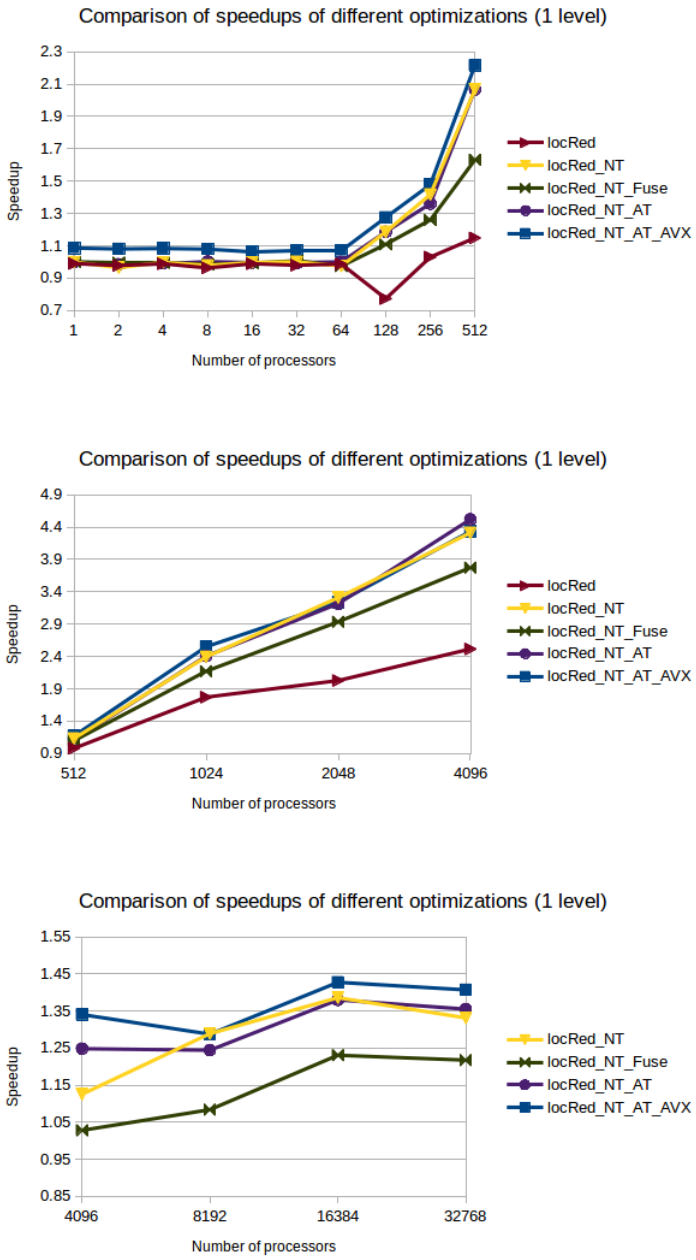


Figure 4.4.: Speedup of the runtime for the different optimization steps compared to the intial version with overlap in test case B. In the range of 4096 to 32768 processors the comparison is to the first optimization (locRed) as no measurements for the initial version are available.

might only be reduced because of less synchronizations in the new version compared to the $MPI\_Iallreduce$ which increases the overlap of computation and communication.

The next step (locRed_NT) is the reduction of the import load and communication partners. First, the NT import was implemented which should be beneficial especially for few local levels as the import volume savings are maximal in these cases (see table 3.1). The strong scaling results in appendix A.2 support this hypothesis. Even for 2 local levels the scaling improves noticeably which shows the benefit of the NT approach. For 1 local level the improvements are larger and the waiting time (appendix A.2) as well as the scaling could be improved so that good scalability until 4096 processors can be achieved. This is also reflected in the speedups relative to the initial version which are significantly improved compared to the first optimization for all test cases. Again test case B with 1 local level shows the largest improvements of this optimization. These observations can also be explained by considering the performance model in [20] with approximated communication time $T_{\alpha-\beta-\gamma} = \alpha + n\beta + (h - h_m)\gamma$ where α is the latency of the system, n the message size in bytes, β the inverse bandwidth, h the number of hops of a specific message and h_m the minimal possible number of hops for a message. In [20] it was observed that for the local tree part the communication time is mostly influenced by the communication volume whereas the communication in the global tree part is dominated by the hop costs. Using the NT import areas a reduction of communications is achieved which at the same time reduces latency α and the total message size n. Furthermore, less communications are performed in the global tree levels. Hence, communication in both tree parts is improved.

Another optimization is the addition of the fused communication to the method (locRed_NT_Fuse). Unfortunately, despite the similar scaling, the method with fused communication showed slightly larger absolute runtimes with no improvement of scalability (see figs. 4.3 and 4.4). The waiting times (see appendix A.2) seem to be significantly improved compared to the previous version, which can be explained by the reduction of communication partners and therefore less latency and overall hops. However, the additional synchronization in the downwardpass, caused by the method, apparently leads to more communication overhead than it saves. This effect is also seen in the relative speedups compared to the initial version which are lower than in the optimization without fused communication. Hence, for the next optimization steps fused communication was disabled.

The auto tuning (locRed_NT_AT) of the starting level of the $MPI\_Iallreduce$ (see section 3.5), improved the runtime especially for small processor numbers where a global reduce is superior to individual sends. This is caused by the MPI optimizations of the reduce operator. Unfortunately, in some cases the auto tuning does not find the best level which leads to slightly longer runtimes (see figs. A.14 and A.15). A possible solution to improve the auto tuning could be to use multiple steps for the comparison of the different levels. In this way, the waiting times which sometimes vary between the different processors and simulation steps, could be predicted more accurately.

Another test was carried out to compare the AVX vectorization (locRed_NT_AT_AVX) to the SSE vectorization (locRed_NT_AT) for the FMM implementation. Previous results

were performed with SSE optimizations although the SuperMUC architecture supports AVX. Therefore, an AVX vectorization was tested to analyse the additional benefit for the runtime. As the P2M and L2P operators are not vectorized, the improvements were limited to about 9% of reduction in the sequential runtime. Therefore, the overall scaling was only affected slightly as seen in figs. 4.5 and 4.6 but a reduction in the absolute computation time was observed. The relative speedup to the initial versions shows that the AVX version performs best for almost all scenarios and processor numbers.

4.3. Final Results

For a more detailed performance analysis the final optimization with AVX vectorization (see figs. 4.5 and 4.6) was further examined. By looking at the runtime of the single operators for 2 levels (see fig. 4.7) it can be observed that the computation of operators in the local subtree scale well with the number of the processors. Unfortunately, the global work increases with the number of processors which will at some point prevent further reductions of the runtime. However, the observed decrease in scalability is mostly caused by the increase in waiting time. Hence, it needs to be investigated if more sophisticated approaches could lower these waiting times even further. Also faster interconnects and memory controllers could be beneficial as they decrease communication time and, therefore, the waiting time. By looking at the other test cases (see appendix A.2) the same observations can be made. The only difference is that for 1 level the waiting times have a larger overall impact compared to the 2 level cases. This is the reason for the worse scaling with 1 local level. It should be noted that $FMMtotal$ in fig. 4.7 also includes the runtime of the P2P, M2P and L2P operators.

Figure 4.8 shows the speedup for 2 local levels in test case A. For larger processor ranges the relative speedup compared to the starting value of the respective range was evaluated. The results show a good speedup of 347 for 512 processors with 5 tree levels, a relative speedup of 5.7 between 512 and 4096 processors for 6 levels and a relative speedup of 3.6 between 4096 and 32768 processors with 7 levels. This is a huge improvement compared to respectively 303.5, 3.3 and 0.7 of the initial version. Consequently, the implementation reaches a parallel efficiency of 68% up to 512, 71% from 512 to 4096 and 46% from 4096 to 32768 processors (see fig. 4.8).

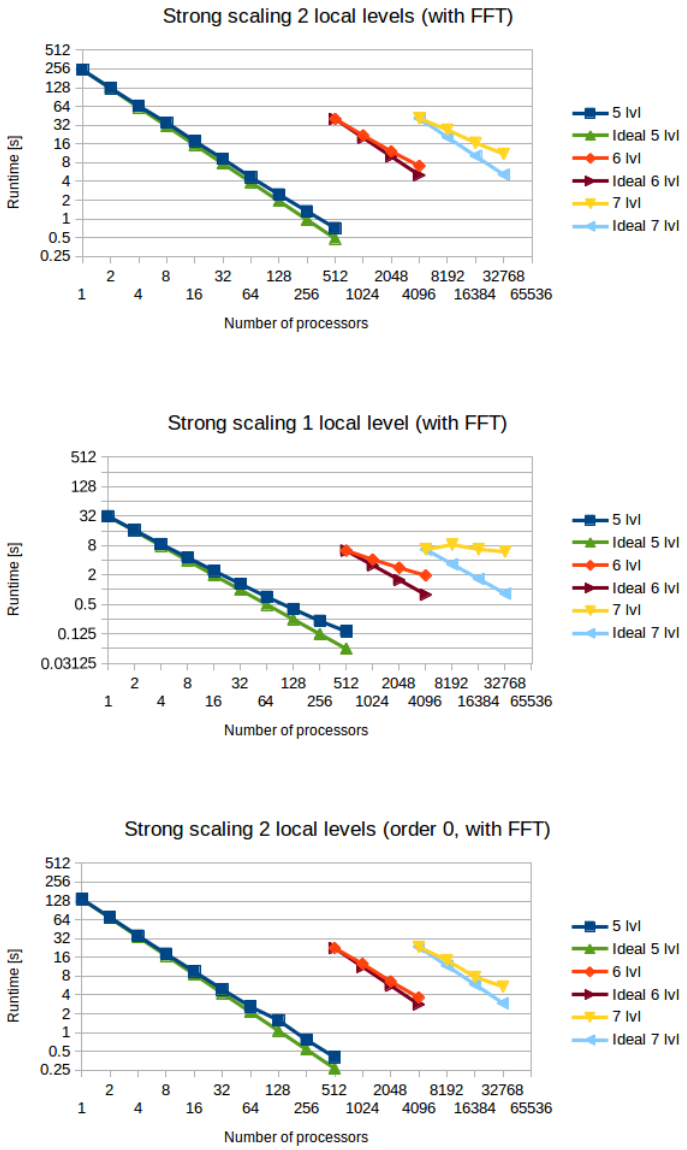


Figure 4.5.: Strong scaling results for test case A (top), B (center) and C (bottom) with FFT acceleration, AVX vectorization, dynamic global reduce adjustment and NT import areas.

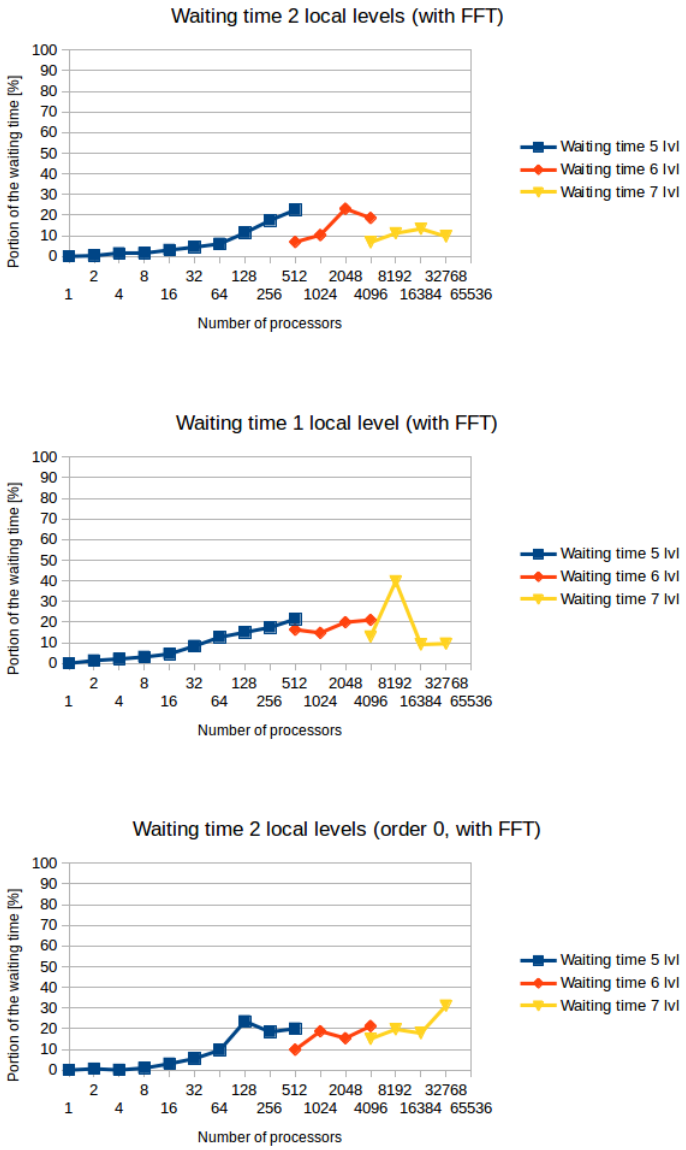


Figure 4.6.: Amount of time waited in test case A (top), B (center) and C (bottom) with FFT acceleration, AVX vectorization, dynamic global reduce adjustment and NT import areas compared to the complete simulation time of the FMM algorithm.

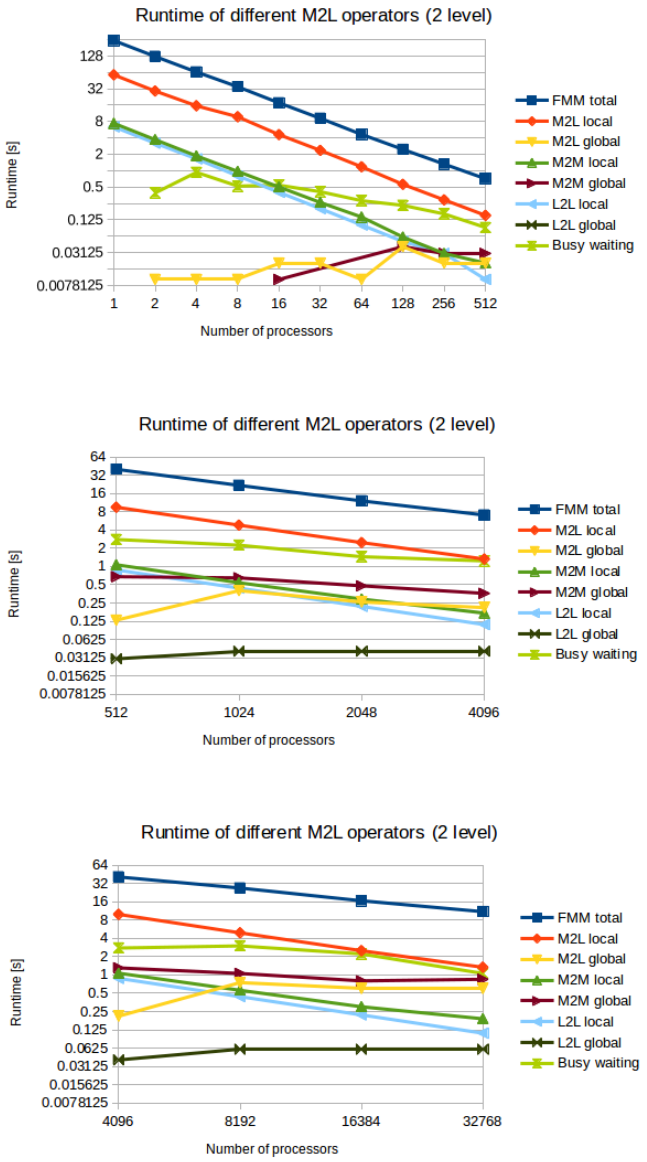


Figure 4.7.: Runtime of single operators during the FMM algorithm compared to the overall runtime and waiting time for test case A. FMM total also includes the runtime of the P2P, M2P and L2P operators.

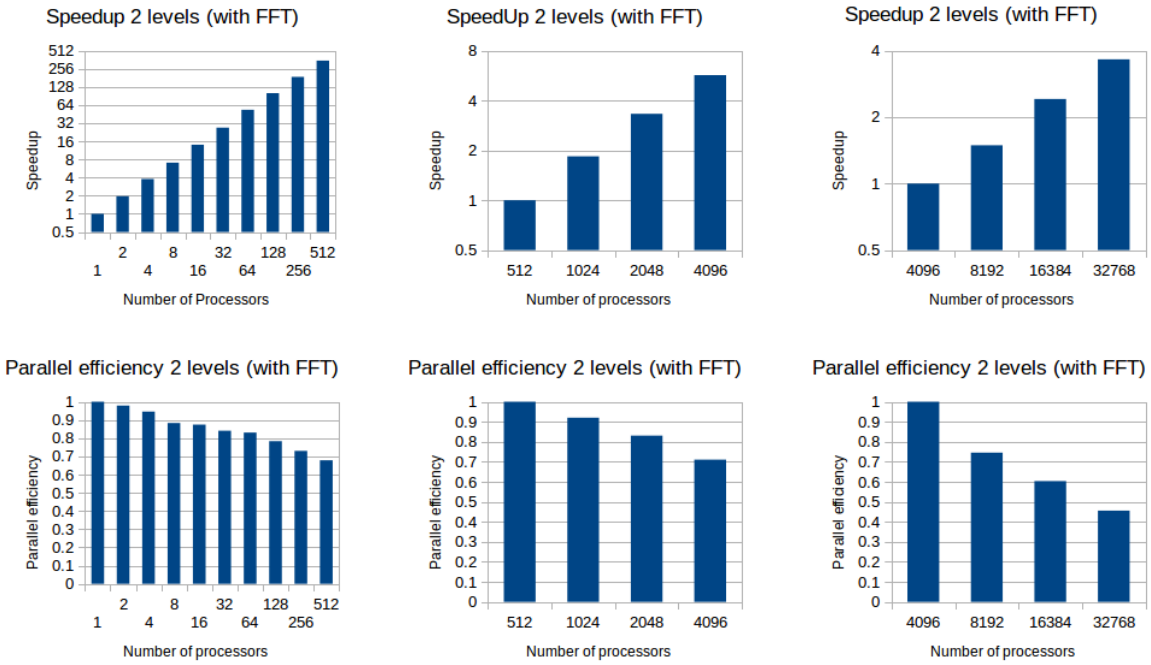


Figure 4.8.: Speedup(top) and parallel efficiency(bottom) for the implementation with NT import regions, dynamic reduce optimization and AVX vectorization in test case A.

Number of particles	672799
Order	5
Tree depth	5 or Ncrit = 21
Periodicity	1 periodic image
Vectorization	AVX
Distribution	Uniform
Parallelization	MPI only

Table 4.4.: This table shows the simulation setup for the comparison of MarDyn and ExaFMM.

4.4. Comparison to ExaFMM

In this section the final implementation is compared to the FMM implementation of ExaFMM [1] (released on 6/5/2016). It should be noted that it is, in general, hard to compare FMM implementations as different expansions - spherical in Mardyn and cartesian in ExaFMM - and different parallel or sequential accelerations are often used. Furthermore, periodic boundary conditions might have different impact on the implementations. Due to these difficulties, comparisons between different FMM codes should be taken with a grain of salt.

As ExaFMM is optimized for low orders of the multipole expansion and MarDyn for larger orders, the order was chosen to be 5 as it should produces similar performance for both methods (see fig. 4.9). As ExaFMM performs best with the Dual Tree traversal (see section 2.1.2), the DTT version is compared to the list-based approach of this work. In order to get a similar acceptance criterion for DTT compared to the list-based approach, $theta$ was set to 0.53 to get a MAC that does not allow for interaction to any cell that is a direct neighbour but allows interaction to any cell which is not a direct neighbour. Here it is assumed that all cells are cubic which is not a necessary precondition for the DTT implementation. For arbitrary sizes it is, however, not possible to construct a $theta$ that gives the same results as the list base traversal for every scenario. The constant $Ncrit$ which controls the depth of the tree was set to the average number of particles in a leaf cell which is approximately 21 in our scenario. Consequently, each approach should produce similar tree depths for both implementations. Particles were placed uniformly in both scenarios and 10 iterations, which are equal to 12 force evaluations, were performed by both implementations. Furthermore, the codes were compiled using AVX vectorization and without mutual force calculation. For parallelization only MPI was used without shared memory parallelization. Table 4.4 shows further parameters like the number of particles.

In fig. 4.10 one can see the runtimes of both implementations. While ExaFMM performs better for small numbers of processors, MarDyn outperforms ExaFMM for large scenarios. This might be caused by the $MPI\_Alltoall$ operation used in the ExaFMM implemenation for the cell exchange. As collective operations tend to produce bad scaling for large processor numbers, a reduction of runtime could only be observed until 128 processors. Another explanation for the worse scaling of ExaFMM is a suboptimal choice of parameters. Other choices for the expansions like spherical expansions or the list-based traversal instead of the dual tree traversal might give better results. Furthermore, the multipole data in ExaFMM scales with $O(p^3)$ which produces more communication load as the $O(p^2)$ complexity of the spherical expansions in MarDyn. In addition, the communication of the halo particles is included in the FMM runtime in contrast to our implementation (see section 2.3). Larger test ranges are omitted as no reduction in runtime could be obtained for ExaFMM.

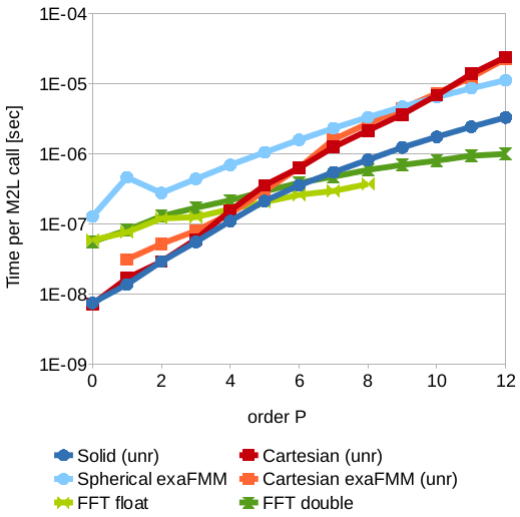


Figure 4.9.: Comparison of the runtime of different M2L implementations depending on the order of the multipole expansion. The M2L time for the cartesian implementation of ExaFMM and our FFT implementation match best at order 5. (image source [28])

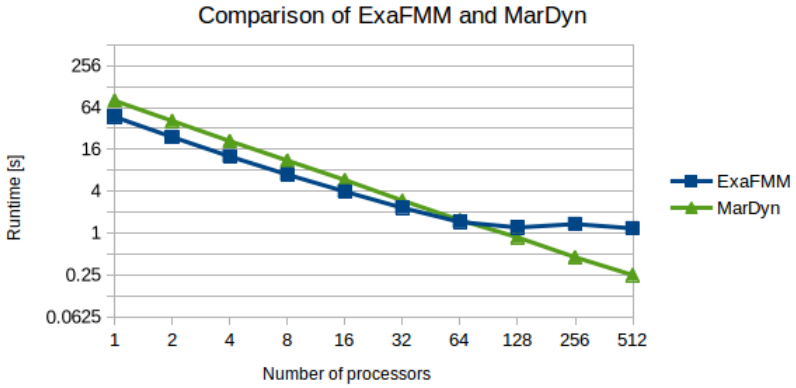


Figure 4.10.: Comparison of the runtime for MarDyn and ExaFMM for different numbers of processors.

5. Conclusion and Outlook

In this work we investigated different approaches for improving the efficiency of a parallel FMM implementation. It was shown that collective operations may reduce scalability drastically. Removing these collectives and reducing import load could significantly improve parallel efficiency and therefore overall computation times for large number of processors. The application of zonal methods has been shown to be beneficial for optimizing communication costs. Therefore, a novel adaptation of the NT method was implemented for FMM which could reduce import loads as well as communication partners and could improve parallel efficiency especially for small test cases. Hence, strong scaling to higher number of processors is possible with such approaches which enables faster computations even at small problem sizes.

Bearing the difficulties of comparing FMM implementations (see section 4.4) in mind, we could outperform ExaFMM in a uniform test case for 128 or more processors. However, for small processor numbers ExaFMM benefits from its highly optimized single-core performance resulting in lower runtimes compared to MarDyn. Hence, improving single-core computation should be one of the main tasks for future research in order to get comparable simulation time as the DTT approach of ExaFMM.

Another important aspect is the application of hybrid parallelization techniques. As modern supercomputers mostly combine shared memory and distributed memory architectures, the usage of OpenMP [6] or other shared memory approaches is possible. Hence, scalability could be improved through the reduction of MPI ranks which causes a decrease of communication and at the same time an increase in local operations. However, special care needs to be taken when it comes to load balancing threads in shared memory approaches as some cell interactions might be filtered due to the well separation criterion or the NT method which only interacts tower with plate cells. Furthermore, border cells compute less if only local interactions are considered during the overlap with the communication. Consequently, sophisticated approaches are necessary to outperform an MPI only implementation which is ideally load balanced for a uniform distribution.

Furthermore, the application of approaches like zonal methods in adaptive scenarios is an important topic. One way to achieve this would be to apply our optimizations to the DTT method. In this way adaptive scenarios can be handled naturally and a broader spectrum of applications could be targeted. Another way would be to use the adaptive FMM method [12] and tailor our approaches to this new setting. One of the biggest issues would be again the task of load balancing in such scenarios. However, it would also enable the usage of processor numbers that are not powers of 2 which might be desirable for some applications and architectures.

Appendix

A. Scaling Results

In this chapter various scaling results are presented for the initial version as well as the different optimizations.

A.1. Scaling of initial version with overlap

Figures A.1 to A.4 show the strong scaling and the waiting times for the initial version with overlap.

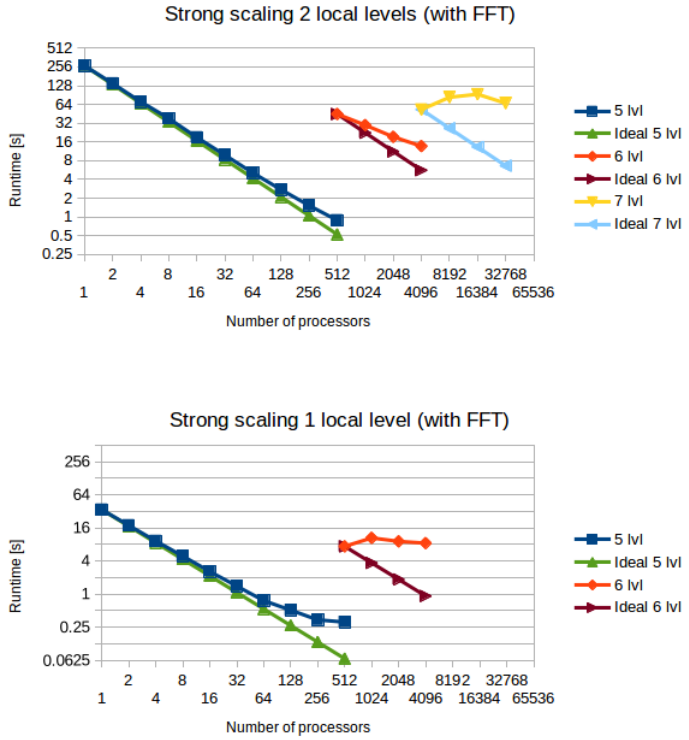


Figure A.1.: Strong scaling results for test case A (top) and B (bottom) with FFT acceleration in the initial parallelization with overlap.

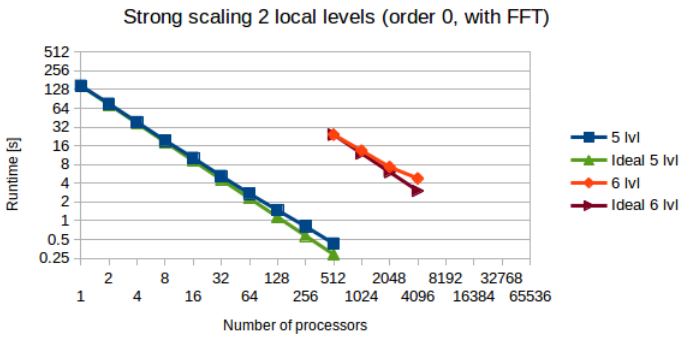


Figure A.2.: Strong scaling results for test case C with FFT acceleration in the initial parallelization with overlap.

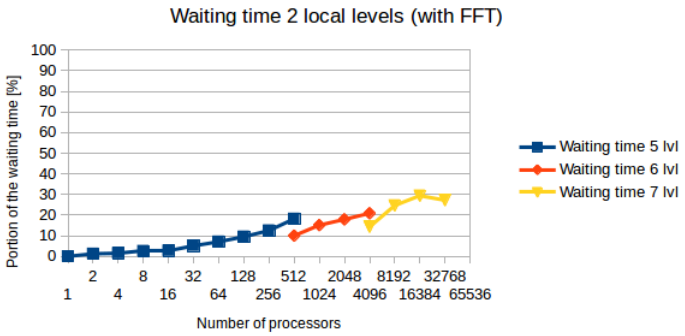


Figure A.3.: Amount of time waited in test case A with FFT acceleration compared to the complete simulation time of the FMM algorithm.

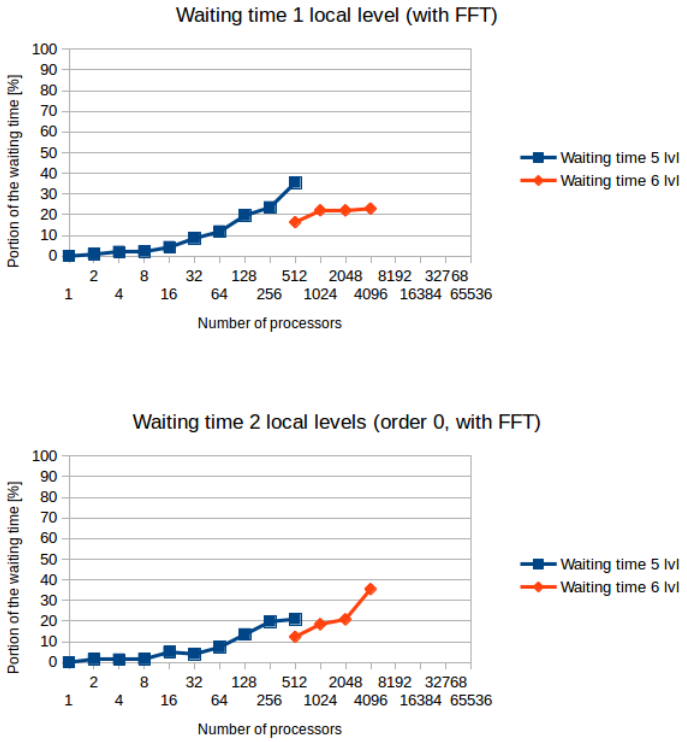


Figure A.4.: Amount of time waited in test case B (top) and C (bottom) with FFT acceleration compared to the complete simulation time of the FMM algorithm.

A.2. Scaling results of the different optimizations

In this section the scaling results of the different optimizations compared to the initial version are presented for varying processor numbers.

Overview of speedups for test case C

Figures A.5 and A.6 shows the speedup of the different optimizations compared to the initial version with overlap. As in the processor range of 4096 to 32768 no initial runtimes were measured, the runtimes are compared to the first optimization (locRed).

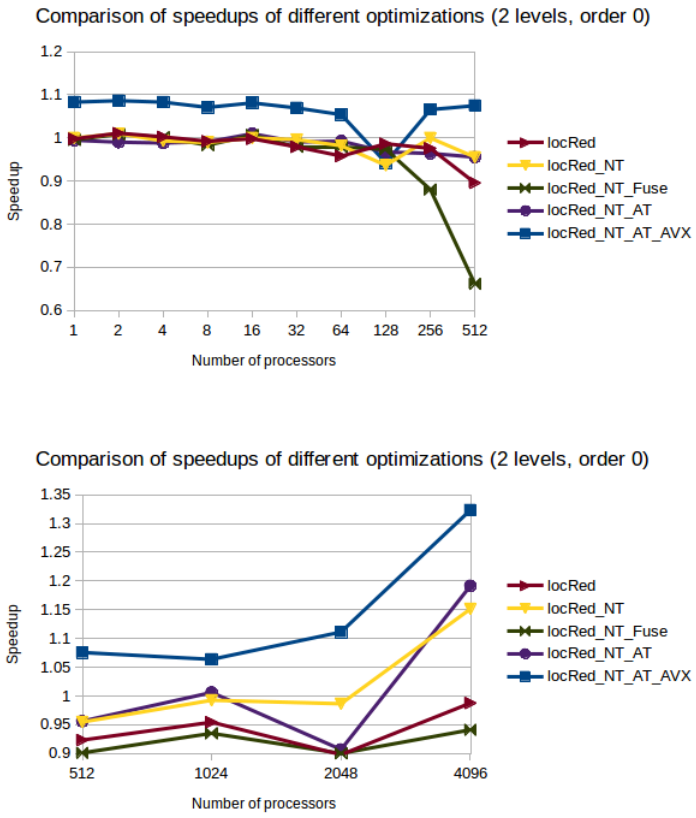


Figure A.5.: Speedup of the runtime for the different optimization steps compared to the intial version with overlap in test case C for 1 to 512 processors.

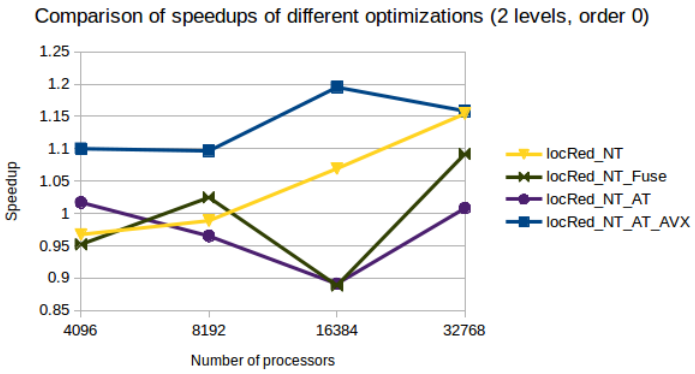


Figure A.6.: Speedup of the runtime for the different optimization steps compared to the first optimization (locRed) in test case C for 4096 to 32768 processors.

Scaling results for optimization locRed

The scaling results and waiting times for locRed, i.e. without global reduce, without NT import loads and without fuse or auto tuning, are show in figs. A.7 to A.9.

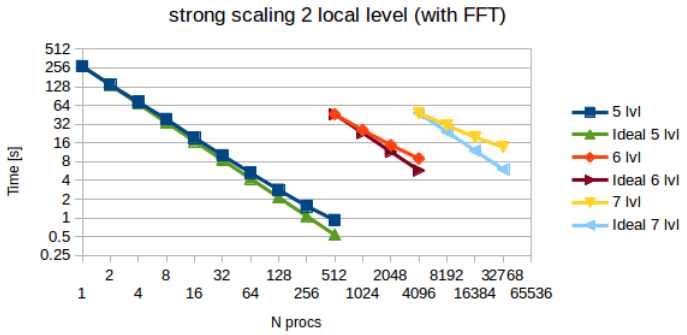


Figure A.7.: Strong scaling results for test case A with FFT acceleration and no global reduce with overlap.

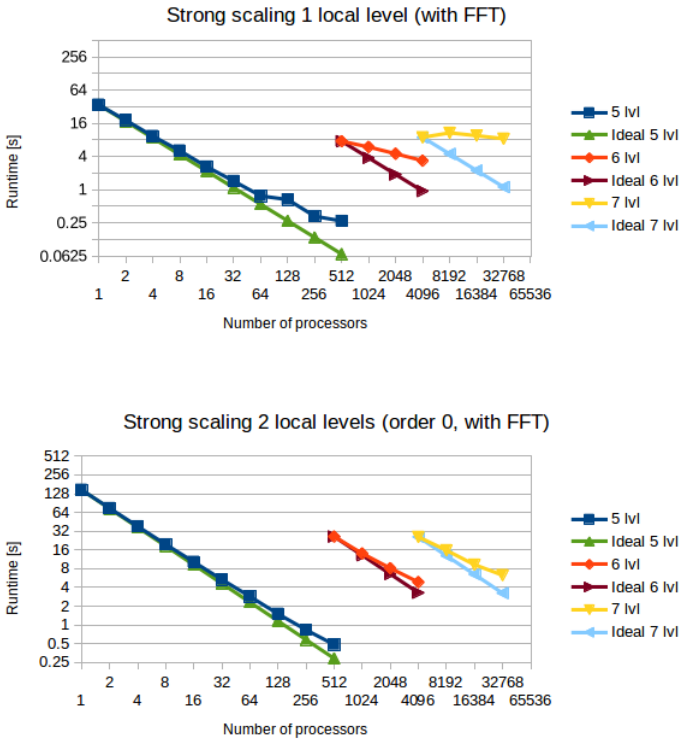


Figure A.8.: Strong scaling results for test case B (top) and C (bottom) with FFT acceleration and no global reduce with overlap.

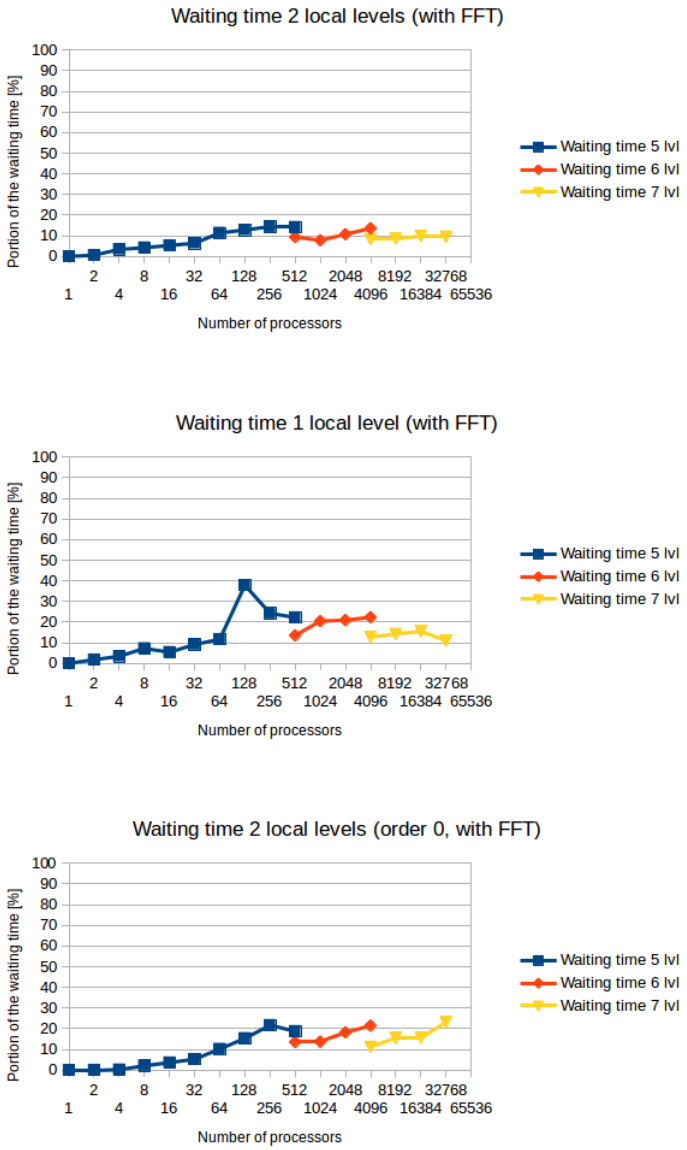


Figure A.9.: Amount of time waited in test case A (top), B (center) and C (bottom) with FFT acceleration and no global reduce compared to the complete simulation time of the FMM algorithm.

Scaling results for optimization locRed_NT

The scaling results and waiting times for locRed_NT, i.e. without global reduce, with NT import loads and without fuse or auto tuning, are shown in figs. A.10 and A.11.

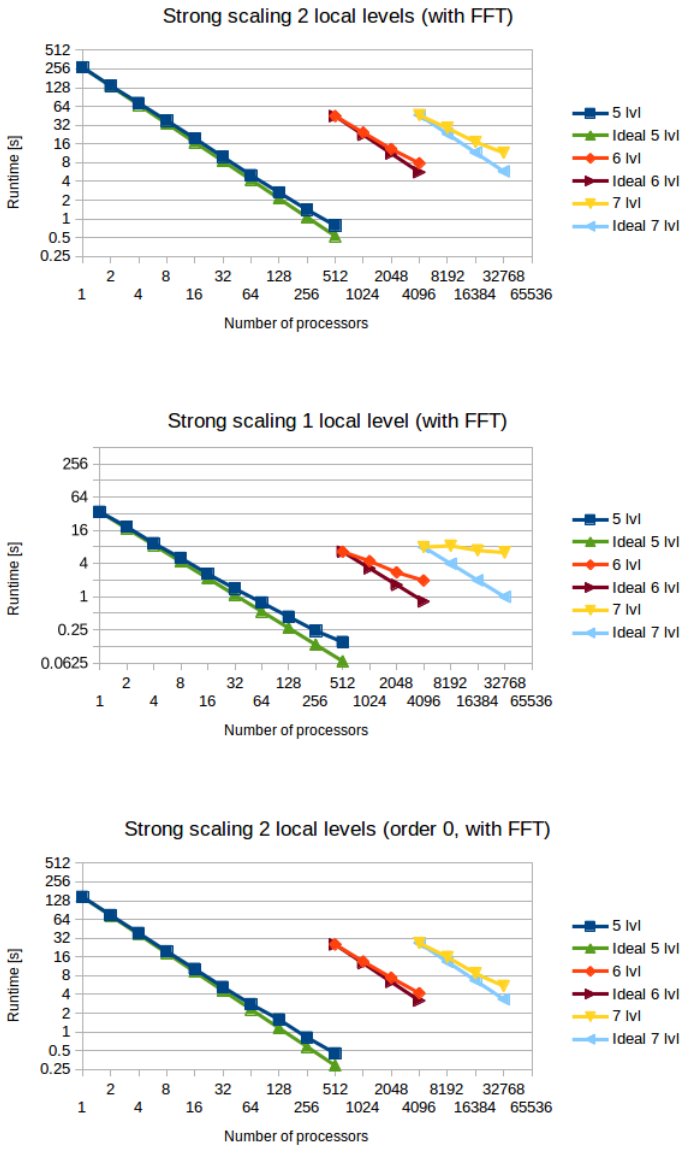


Figure A.10.: Strong scaling results for test case A (top), B (center) and C (bottom) with FFT acceleration, no global reduce and NT import areas.

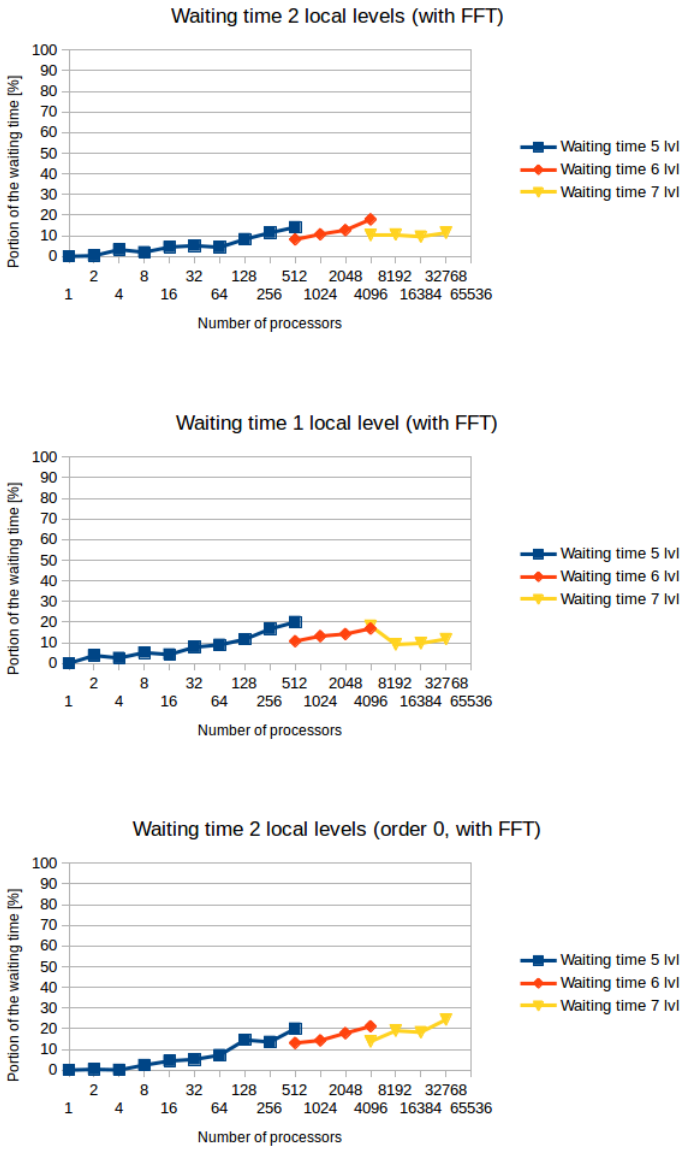


Figure A.11.: Amount of time waited in test case A (top), B (center) and C (bottom) with FFT acceleration, no global reduce and NT import areas compared to the complete simulation time of the FMM algorithm.

Scaling results for optimization locRed_NT_Fuse

The scalings results and waiting times for locRed_NT_Fuse, i.e. without global reduce, with NT import loads, with fuse and no auto tuning, are shown in figs. A.12 and A.13.

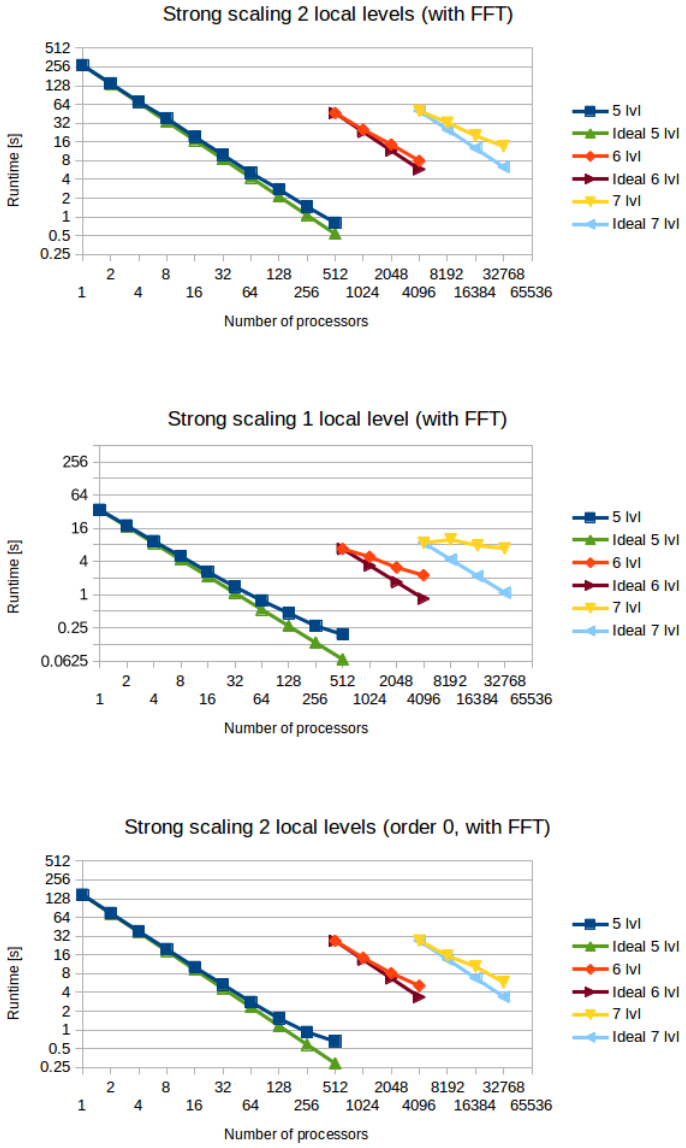


Figure A.12.: Strong scaling results for test case A (top), B (center) and C (bottom) with FFT acceleration, no global reduce, NT import and fused communication.

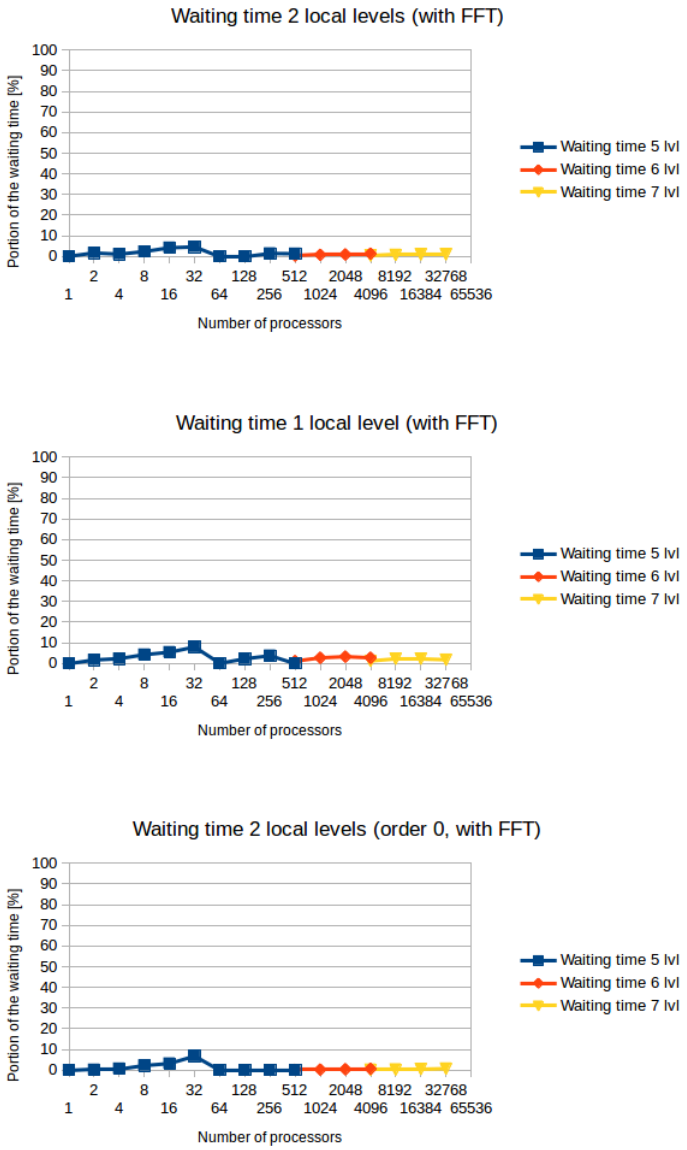


Figure A.13.: Amount of time waited in test case A (top), B (center) and C (bottom) with FFT acceleration, no global reduce, NT import areas and fused communication compared to the complete simulation time of the FMM algorithm.

Scaling results for optimization locRed_NT_AT

The scaling results and waiting times for locRed_NT_AT, i.e. without global reduce, with NT import loads, without fuse, with SSE vectorization and with auto tuning, are shown in figs. A.14 and A.15.

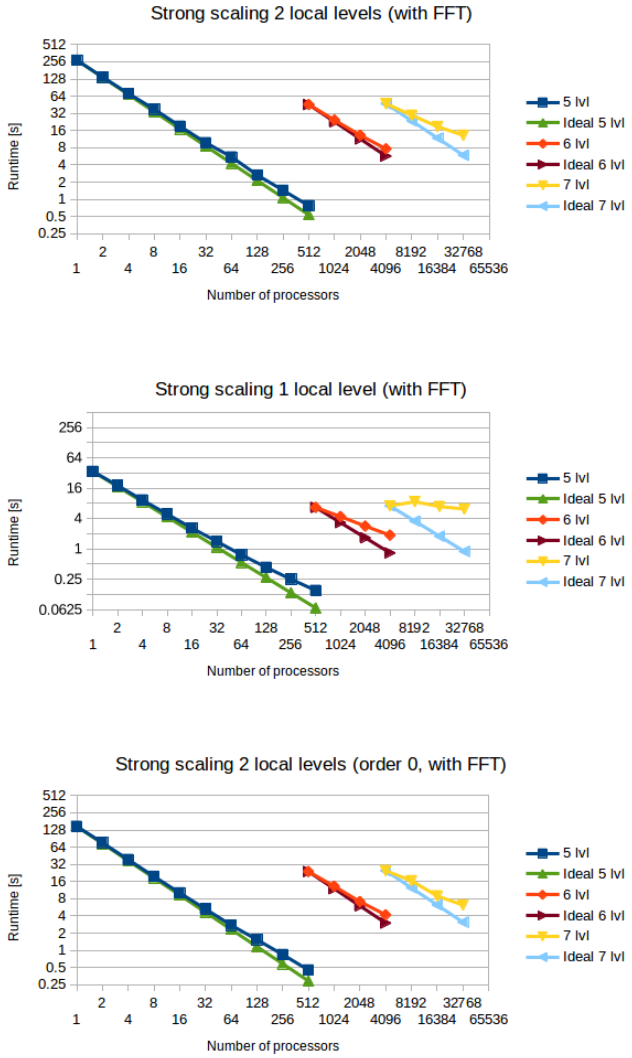


Figure A.14.: Strong scaling results for test case A (top), B (center) and C (bottom) with FFT acceleration, dynamic global reduce adjustment and NT import areas.

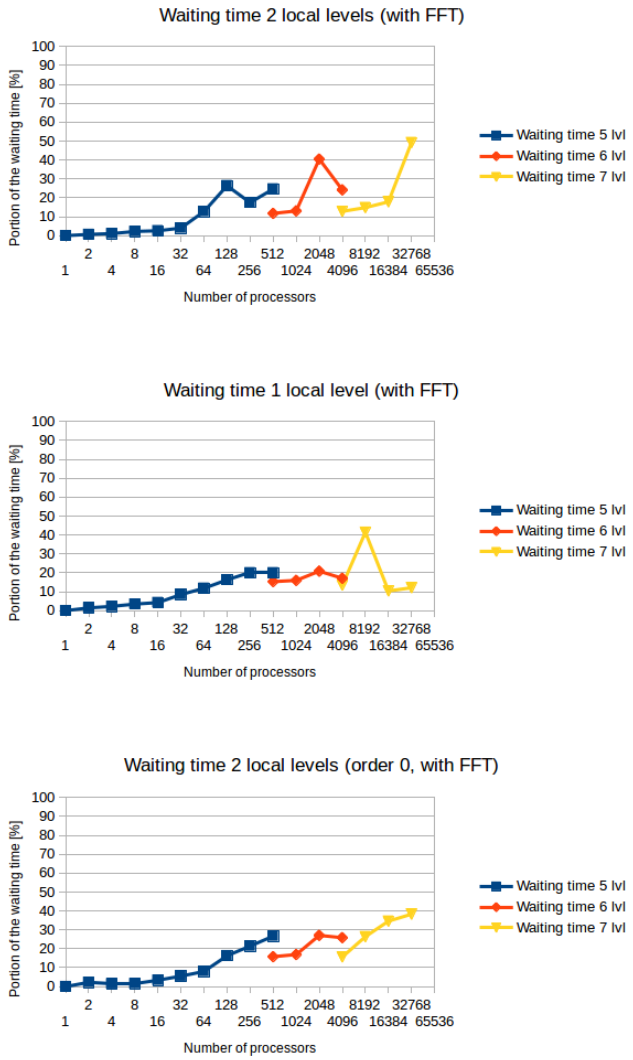


Figure A.15.: Amount of time waited in test case A (top), B (center) and C (bottom) with FFT acceleration, dynamic global reduce adjustment and NT import areas compared to the complete simulation time of the FMM algorithm.

Runtimes of single Fast Multipole operators

In this section the runtimes of the different FMM operators are shown for test cases B and C (see figs. A.16 and A.17).

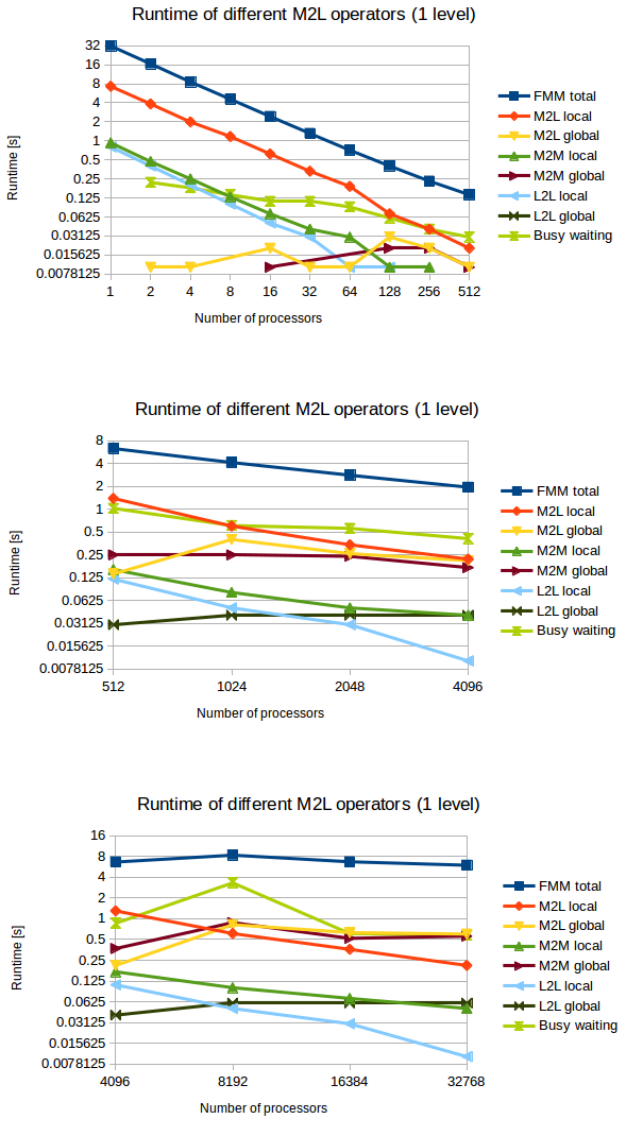


Figure A.16.: Runtime of single operators during the FMM algorithm compared to the overall runtime and waiting time for test case B. FMM total also includes the runtime of the P2P, M2P and L2P operators.

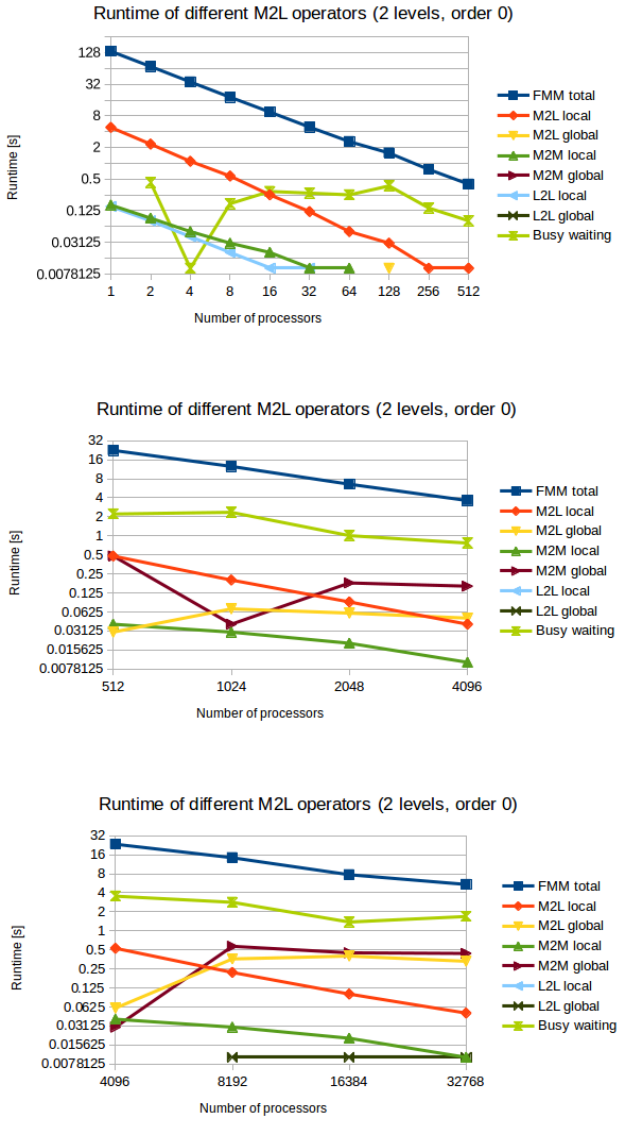


Figure A.17.: Runtime of single operators during the FMM algorithm compared to the overall runtime and waiting time for test case C. FMM total also includes the runtime of the P2P, M2P and L2P operators.

Bibliography

[1] exafmm. https://github.com/exafmm (accessed on 04.10.2016).

[2] Ibm spectrum mpi. http://www-03.ibm.com/systems/spectrum-computing/products/mpi/ (accessed on 25.09.2016).

[3] Intel® mpi library. https://software.intel.com/en-us/intel-mpi-library/documentation (accessed on 25.09.2016).

[4] Top 500: The list. https://www.top500.org/ (accessed on 25.09.2016).

[5] Open mpi: Open source high performance computing. https://www.open-mpi.org/ (accessed on 25.09.2016), September 2016.

[6] Openmp: The openmp® api specification for parallel programming. http://openmp.org/wp/ (accessed on 25.09.2016), August 2016.

[7] Supermuc petascale system. http://www.lrz.de/services/compute/supermuc/, March 2016. (accessed on 22.09.2016).

[8] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[9] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[10] K. J. Bowers, R. O. Dror, and D. E. Shaw. Zonal methods for the parallel execution of range-limited n-body simulations. *Journal of Computational Physics*, 221(1):303–329, January 2007.

[11] M. Buchholz, H.-J. Bungartz, and J. Vrabec. Software design for a highly parallel molecular dynamics simulation framework in chemical engineering. *Journal of Computational Science*, 2(2):124 – 129, 2011. Simulation Software for Supercomputers.

[12] H. Cheng, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm in three dimensions. *Journal of Computational Physics*, 155(2):468 – 498, 1999.

[13] W. Dehnen. A hierarchical o(n) force calculation algorithm. *Journal of Computational Physics*, 179(1):27 – 42, 2002.

[14] J. Dongarra and F. Sullivan. Guest editors' introduction: The top 10 algorithms. *Computing in Science & Engineering*, 2(1):22–23, 2000.

[15] W. Eckhardt, A. Heinecke, R. Bader, M. Brehm, N. Hammer, H. Huber, H.-G. Kleinhenz, J. Vrabec, H. Hasse, M. Horsch, M. Bernreuther, C. W. Glass, C. Niethammer, A. Bode, and H.-J. Bungartz. *591 TFLOPS Multi-trillion Particles Simulation on SuperMUC*, volume 7905 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013.

[16] Message Passing Interface Forum. Mpi: A message-passing interface standard version 3.1, June 2015.

[17] J.-M. Gallard. Optimization, implementation and evaluation of the fft-accelerated fast multipole method. Master's thesis, Technische Universität München, 2016.

[18] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325 – 348, 1987.

[19] M. Griebel, S. Knapek, and G. Zumbusch. *Numerical Simulation in Molecular Dynamics*. Springer Verlag, 2007.

[20] H. Ibeid, R. Yokota, and D. Keyes. A performance model for the communication in fast multipole methods on HPC platforms. *CoRR*, abs/1405.6362, 2014.

[21] I. Kabadshow. *Periodic Boundary Conditions and the Error-Controlled Fast Multipole Method*. Dr. (univ.), Univ. Wuppertal, Jülich, 2012. Record converted from JUWEL: 18.07.2013; Wuppertal, Univ., Diss., 2012.

[22] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.-A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. *Commun. ACM*, 55(5):101–109, May 2012.

[23] 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. *Journal of Chemical Theory and Computation*, 10(10):4455–4464, 2014.

[24] V. Rokhlin. Rapid solution of integral equations of classical potential theory. *Journal of Computational Physics*, 60(2):187 – 207, 1985.

[25] D. E. Shaw. A fast, scalable method for the parallel evaluation of distance-limited pairwise particle interactions. *Journal of Computational Chemistry*, 26(13):1318–1328, 2005.

[26] M. Snir. A note on n-body computations with cutoffs. *Theory of Computing Systems*, 37(2):295–318, 2004.

[27] W. C. Swope, H. C. Andersen, P. H. Berens, and K. R. Wilson. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *The Journal of Chemical Physics*, 76(1):637–649, 1982.

[28] N. Tchipev, J.-M. Gallard, P. Neumann, and H.-J. Bungartz. Comparison of solid harmonics and cartesian taylor expansions for the fast multipole method for laplacian potentials. *5th European Seminar on Computing*, 2016.

[29] Abdulnour Y. Toukmaji and J. A. Board. Ewald summation techniques in perspective: a survey. *Computer Physics Communications*, 95(2):73 – 92, 1996.

[30] L. Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159:98–103, Jul 1967.

[31] M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 12–21, New York, NY, USA, 1993. ACM.

[32] C. A. White and M. Head-Gordon. Rotating around the quartic angular momentum barrier in fast multipole method calculations. *The Journal of Chemical Physics*, 105(12):5061–5067, 1996.

[33] R. Yokota. An FMM based on dual tree traversal for many-core architectures. *CoRR*, abs/1209.3516, 2012.

[34] R. Yokota and L. Barba. Hierarchical n-body simulations with autotuning for heterogeneous systems. *Computing in Science Engineering*, 14(3):30–39, May 2012.

[35] R. Yokota, G. Turkiyyah, and D. Keyes. Communication complexity of the fast multipole method and its algebraic variants. *CoRR*, abs/1406.1974, 2014.