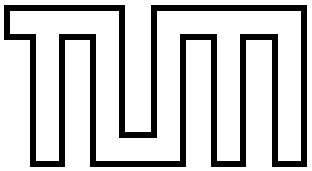


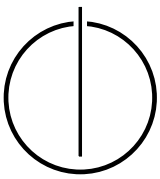
FAKULTÄT FÜR INFORMATIK

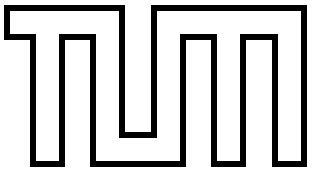
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Task Based Parallelization of the Fast Multipole Method implementation of ls1-mardyn via QuickSched

Fabio Alexander Gratl





FAKULTÄT FÜR INFORMATIK

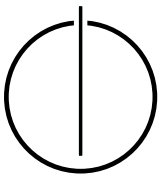
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Task Based Parallelization of the Fast Multipole Method implementation of ls1-mardyn via QuickSched

Task-basierte Parallelisierung der Implementierung der Fast Multipole Methode von ls1-mardyn via QuickSched

Author:	Fabio Alexander Gratl
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Nikola Tchipev, M.Sc.
Date:	November 15, 2017



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

Munich, November 15, 2017 Fabio Alexander Gratl

Acknowledgements

At this point, I want to express my gratitude to the people who supported me throughout this thesis. First of all to the chair for scientific computing and Professor Hans-Joachim Bungartz for offering me the possibility for this thesis and providing me access to all the computing platforms used. Also, a big thank you goes to Nikola Tchipev for his guidance, patient explanations, inspiring discussions, and constructive feedback over the course of this project. Last but not least, I want to thank Josefine for her continuous support and helpful input all around the clock.

Abstract

The Fast Multipole Method is an algorithm for computing long-range interactions in $N-$body problems in linear computational complexity. Since it consists of many individual parts per time step, even optimized fork-join approaches using *OpenMP* carry a significant synchronization overhead [AMP+13]. However, these parts do not need to be executed completely after each other, instead, an interweaving is possible. Therefore, task based approaches with a dynamic dependency model are good candidates for parallelization.

This thesis describes a task based, shared memory parallelization of the implementation of the Fast Multipole Method in the large-scale molecular dynamics code *ls1-mardyn* [NBB+14][Eck14]. Since the approach aims for a maximal scheduling flexibility, the *QuickSched* library was chosen to create and execute tasks as explicit tasks provided by *OpenMP 4.5* are not dynamic enough to model the required dependencies [Gal16].

The approach is tested with a range of parameter configurations and on two different architectures, namely *Intel Ivy Bridge* and the new state-of-the-art *Intel Xeon Phi Knights Landing*.

Through a detailed analysis of the scheduling and scaling behavior it is shown that the here presented approach can achieve good parallel performance, but is highly dependent on a good choice of parameters for the Fast Multipole Method.

Zusammenfassung

Die Fast Multipole Methode ist ein Algorithmus zur Berechnung der Interaktionen innerhalb eines $N-$Körper-Problems über große Distanzen in linearer Rechenkomplexität. Da der Algorithmus pro Zeitschritt aus vielen Einzelschritten besteht, bringen auch optimierte fork-join Ansätze, welche *OpenMP* nutzen, einen signifikanten Mehraufwand durch die erforderliche Synchronisation [AMP+13]. Da diese Einzelschritte jedoch nicht streng nacheinander ausgeführt werden müssen, ist es möglich diese ineinander zu verflechten. Daher sind task-basierte Ansätze mit dynamischen Abhängigkeiten gute Kandidaten für eine Parallelisierung.

Die vorliegende Arbeit beschreibt eine task-basierte Parallelisierung mit geteiltem Speicher für die Implementierung der Fast Multipole Methode im Molekulardynamikcode *ls1-mardyn* [NBB+14][Eck14]. Weil der Ansatz auf maximale Flexibilität während des Schedulings abzielt, wurde die *QuickSched* Bibliothek gewählt, da explizite Tasks von *OpenMP 4.5* nicht dynamisch genug sind um die erforderlichen Abhängigkeiten abzubilden [Gal16].

Der Ansatz wird mit einer Reihe von Parameterkonfigurationen auf der *Intel Ivy Bridge* sowie der modernen *Intel Xeon Phi Knights Landing* Architektur getestet.

Durch eine detaillierte Analyse des Scheduling- und Skalierungsverhaltens wird gezeigt, dass der vorgestellte Ansatz gute parallele Leistung erzielen kann, diese aber jedoch stark abhängig ist von einer guten Wahl der Parameter für die Fast Multipole Methode.

Contents

Part I.

Introduction and Background

1. Introduction

Large-scale molecular dynamics simulations are nowadays able to replicate the behavior of trillions of molecules to on a realistic level [EHB+13]. Therefore, they have become highly useful tools in different fields of science like molecular biology [KDFB04], thermodynamics [DES+11], chemistry [vGB90], or material studies [Bin95]. They can be employed to analyze or predict properties and processes on a molecular level under exactly specifiable and reproducible parameters.

However, since molecular dynamics simulations determine the behavior of a particle by evaluating its interactions with all other particles, they are by their definition, an $N-$body problem. Naively, this class of problems has a computational complexity of $O(N^2)$ since every particle interacts with all others. While efficient algorithms for short-range interactions have are already been established for some time [Pli95], the same cannot be said for long-range interactions. Approaches with good asymptotical computational complexity have been developed, like the Ewald-Summation, which lies in $O(N^{\frac{3}{2}})$, or its further development, the P3M, lying in $O(N \log(N))$. Also, tree-based approaches like the Fast Multipole Method with a complexity of $O(N)$ were published. Yet, asymptotical complexity is not the only thing to consider in practice, which is why especially the Fast Multipole Method was and still is subject to critical analysis. For example, [PG96] compares the P3M, Fast Multipole Method, and Ewald-Summation, concluding:

> "The FMM is a second choice for all system sizes both in terms of speed and program complexity"

Ten years later, [KP06] arrives at the conclusion:

> "[...] the FMM is commonly overestimated in its complexity and underestimated in its performance."

With the desire to simulate ever-growing numbers of molecules the need for linearly scaling algorithms rises even further.

Although Moore's Law, which can be translated to the speed of new CPUs doubling approximately every two years, is still in effect, it is not anymore due to an increase in frequencies, but by the fact that more and more compute cores are fitted on modern processors. Over the last years processors containing 64 cores and more are becoming more and more available and are employed in supercomputers. Most prominently, processors with high core counts are used, the number one of the top 500[1], the Sunway TaihuLight at the National Supercomputing Center in Wuxi[2].

This trend requires a paradigm shift in code and software design to more parallel algorithms. Also, when greater amounts of memory are needed, as it is the case in large-scale simulations,

[1] `https://www.top500.org`
[2] `http://www.nsccwx.cn`

shared memory parallelizations should be considered to avoid duplication of data and overhead by communication.

For these reasons, this thesis explores the potential of a shared memory parallelization of the Fast Multipole Method in the molecular dynamics simulation *ls1-mardyn*.

2. Theoretical Background

Molecular dynamics simulations bring together the fields of molecular chemistry and computer science. Here, the necessary theoretical background shall be explained. This covers the underlying physical processes on the one hand and the applied algorithms that allow for efficient computation of the problem on the other hand.

2.1. Intermolecular Potentials

Before discussing the Fast Multipole Method, the relevant intermolecular potentials need to be defined. Here, only pairwise potentials are considered and their effects are assumed to be exactly additive. Those potentials can be divided into short- and long-range potentials depending on how their influence decreases on with growing distance between interacting particles.

2.1.1. Lennard-Jones Potential

The Lennard-Jones Potential U_{LJ}, also-called the L-J Potential or 12-6 Potential, was first proposed in 1924 by John Lennard-Jones and John Edward [LJ24]. It is a very simple mathematical model for attracting forces, like van der Waals or London dispersion forces, and repulsive forces, like Pauli repulsion, between two particles. According to these forces, particles induce a repulsive potential, which declines with spacial distance and can be seen in Figure 2.1. When a certain distance ϵ is reached, the potential turns into an attractive force growing with further distance up to a value *sigma*. After that, the potential converges to zero with more spacial separation[LJ31].

$$U_{LJ}(P_1, P_2) = 4\epsilon \left(\left(\frac{\sigma}{R} \right)^{12} - \left(\frac{\sigma}{R} \right)^{6} \right) \tag{2.1}$$

In Equation 2.1, R is the distance between the particles P_1 and P_2. The ϵ represents the minimal potential between the particles and σ the spacial distance where attractive and repulsive forces cancel out exactly.

For small R the minuend is the dominant part of the term due to its larger exponent, which is why it can be seen as the repulsive term. Conversely, the subtrahend is dominant for larger R, what makes it the attractive term. A visualization of this can be seen in Figure 2.1 for $\epsilon = 1$ and $\sigma = 1$.

Due to its computational simplicity and fairly good approximation of interparticle forces, it is often used in molecular dynamic simulations [GKZ07].

Since molecules consisting of several particles become more complex, several so-called Lennard-Jones centers are considered per molecule, which are points in the molecule from where the potential is calculated. When computing the Lennard-Jones potential between two

two-centered molecules, all pairwise interactions except the interactions within the molecules need to be considered. An example of this situation is depicted in Figure 2.3. An increased number of Lennard-Jones centers exponentially increases the number of calculations necessary since for n molecules with m centers each $O(n^m)$ computations need to be executed.

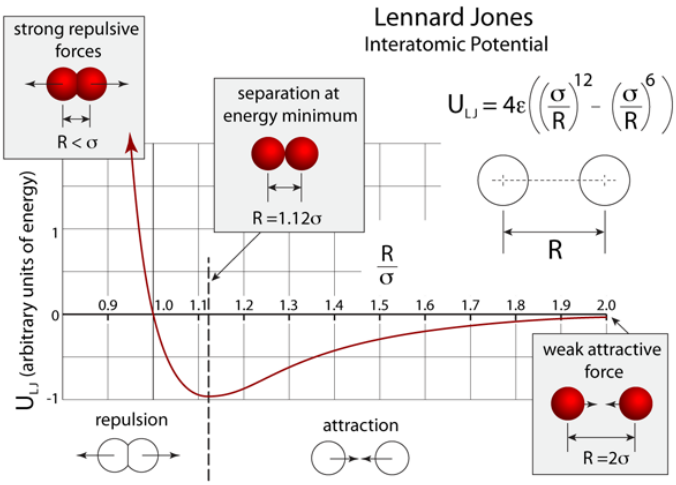


Figure 2.1.: Profile of the Lennard-Jones Potential for $\epsilon = 1$ and $\sigma = 1$.
Source: http://atomsinmotion.com/book/chapter5/md

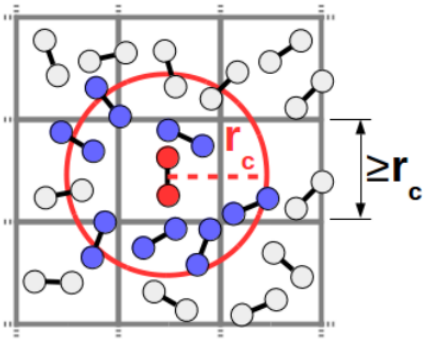


Figure 2.2.: Cutoff radius around a molecule (red circle). Only blue molecules are considered for force calculation.
Source: [TWG+15]

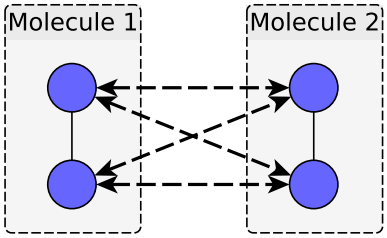


Figure 2.3.: Interactions between multi-centered molecules. Blue discs represent Lennard-Jones centers, dashed arrows interactions.
Source: [Gra17]

2.1.2. Electrostatic Potential

The Coulomb Potential is one of the fundamentals of electrostatics. It was discovered by Charles Augustin de Coulomb in 1785 [Cou85].

$$U_C(q_1, q_2) = \frac{1}{4\pi\epsilon_0} \cdot \frac{q_1 \cdot q_2}{R} \tag{2.2}$$

ϵ_0 in the first fraction of Equation 2.2 is the so-called vacuum permittivity and the whole fraction is called *Coulomb's constant*, which roughly evaluates to $8.99 \cdot 10^9 N m^2 C^{-2}$. Both

q_1 and q_2 are point charges or charges spread over separated spheres and R is the distance between their centers.

There is also the possibility of several charges distributed over one molecule, the most common example being a water molecule [Ran09]. The equations for how the influence of these dipoles or quadrupoles can be calculated are stated in [Eck14] or more in-depth in [GGJ11].

2.2. Linked-Cell Algorithm

In Chapter 1 it was mentioned that the calculation of forces in molecular dynamic simulations is an $N-$body problem since theoretically, every particle induces a potential on every other one. Thereby, the naive direct computation results in a computational complexity of $O(N^2)$, which is not to be desired.

As explained in Subsection 2.1.1, the attractive part of the Lennard-Jones Potential decays rapidly with increasing distance between the interacting particles because of its sixth power. This results in a negligible potential between particles which are far apart. Therefore, the Lennard-Jones Potential can be considered a near-field force.

This property is exploited by the Linked-Cell algorithm, which reduces the particles to be considered for each particle only to those in a fixed cutoff radius r_c. Beyond this, the potential between the particles is sufficiently small to not induce a significant error.

For ease of addressing the single molecules, the domain is subdivided into cells. These do not necessarily need to be of uniform size but for the sake of simplicity and to be in line with the current implementation of *ls1-mardyn* a uniform cell size is assumed. The length of these cells is typically r_c, so only all neighbor cells, including diagonal neighbors, need to be searched when looking for particles inside of the cutoff range. This is visualized in Figure 2.2. Smaller cell sizes lead to more cells to be considered but also to less unnecessary distance calculations as the finer grid better approximates the cutoff radius [Eck14].

Since only a constant number of cells needs to be considered for each cell and the number of cells is typically chosen proportional to the number of particles N, the computational complexity of the Linked-Cell Algorithm lies in $O(N)$.

2.3. Fast Multipole Method

Not all potentials decay as fast over distance as the Lennard-Jones potential and must therefore also be taken into account for distant particles. An example is the Coulomb potential, which was explained in Subsection 2.1.2.

Calculating the potential between all particles directly would naively again result into a problem of complexity in $O(N^2)$, whereby N is the total number of particles in the system. In order to tackle this, Leslie Greengard and Vladimir Rokhlin in 1987 proposed the Fast Multipole Method [GR87], which applies certain simplifications reducing the complexity down to $O(N)$.

Another notable approach is the Ewald-Summation, the oldest algorithm for efficiently calculating long-range potentials, which was proposed in 1921 by Paul Peter Ewald [Ewa21]. It works on the idea of splitting potentials into a short-range and a long-range contribution by introducing a cutoff radius. The short-range contribution can be calculated in linear time,

for example by the Linked-Cell Algorithm. Since the long-range contribution is a smooth function it can be evaluated in Fourier space by only looking at the first few coefficients of the transform as these low-frequency components dominate [Eck14]. It is also important to acknowledge that by using the Fourier transform, an infinite number of periodic images is considered implicitly, which leads to a very high accuracy of this method. However, as shown in [Fin94], the optimal computational complexity for the Ewald-Summation is $O(N^{\frac{3}{2}})$, which makes it suboptimal for huge simulations.

Commonly employed improvements of the Ewald-Summation are the Particle-Particle-Particle-Mesh (P3M) [EHL80] and later the Particle-Mesh-Ewald (PME) [DYP93]. Both apply interpolations to obtain a grid for the charges and then perform fast Fourier transforms. This results in an improved complexity of $O(N \log(N))$.

A detailed comparison of accuracy and actual run-time between the Ewald-Summation, the Particle-Mesh-Ewald, and the Fast-Multipole-Method was made in [Pet95]. It is concluded that starting from about 10^5 particles the Fast-Multipole-Method seems to be faster but the Particle-Mesh-Ewald's accuracy prevails. However, it needs to be noted that this study is from 1995 when the Fast-Multipole-Method was rather new. Also, various choices in the implementation make it difficult to compare different Fast-Multipole-Method codes. In [KP06] a more recent analysis of such choices is made. Furthermore, the authors argue, that the Fast Multipole Method is also a good choice for long simulations of smaller systems.

2.3.1. Intuition

The first idea of the Fast Multipole Method is to split the contributing long-range forces in near- and far-field interactions depending on the spacial distance between the source and target particles.

$$\Phi = \Phi_{near} + \Phi_{far} \tag{2.3}$$

Near-field interactions Φ_{near} need to be calculated directly, for example by means of the Linked-Cell Algorithm described in Section 2.2 in $O(N)$ by defining a cutoff radius.

Far-field interactions Φ_{far} work on the idea that clustered particles can be combined into one virtual pseudo-particle. This can be done by using the multipole and local expansion which are derived in [GR87] with spherical harmonics, and for the three-dimensional case in [GR88]. In order to computationally simplify the calculations, the algorithm can also be formulated using solid harmonics, which are derived from the spherical harmonics, which is explained in [Ell95].

Figure 2.4 visualizes how the multipole and local expansion approximate the exact potential. The particles p_1 and p_2 induce potentials at $r = 1$ and $r = -1$, respectively, which is depicted by the green graph. This is the potential that needs to be approximated. Using the multipole expansion, a pseudo-particle m is created at $r = 0$, which approximates the two previous particles. Its potential is represented by the red graph. However, it must be considered that the quality of the approximation by the red graph for the green one increases with distance to m and is only sufficiently good after a certain distance from the pseudo-particle. This means, that the multipole expansion is only a good approximation of the real potential for interactions from a certain distance. The converse is true for the local expansion, which is depicted by the blue line. It is only a good approximation for the direct vicinity of the pseudo-particle.

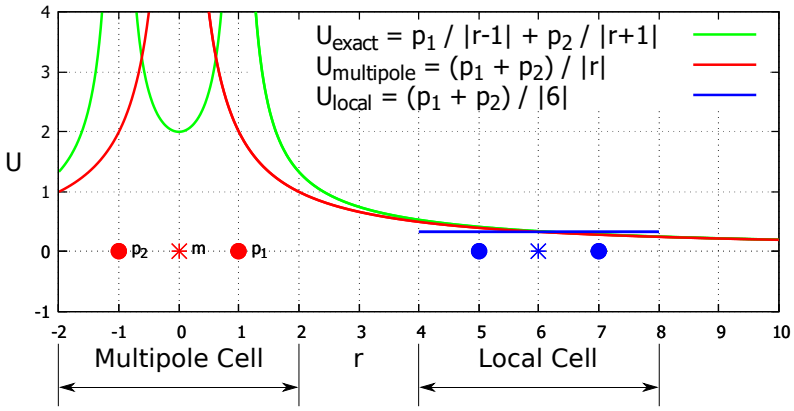


Figure 2.4.: Profiles of the exact potential (green), the approximation by the multipole expansion (red), and by the local expansion (blue) relative to spacial distance r.
Source: Based on a similar figure by Nikola Tchipev.

Using these expansions, a tree can be constructed by first grouping particles to multipole expansions and then group those expansions to larger multipole expansions. In the two-dimensional case, this results in a quadtree or an octree in three-dimensions. To compute interactions between the nodes of the tree, special operators need to be used. Since this thesis is mainly concerned with the parallelization of the Fast Multipole Method its mathematical details and the definition of the operators will not be discussed. The interested reader is therefore redirected to the sources mentioned above and especially to [Gal16], whose description matches the here used variation of the Fast Multipole Method best.

2.3.2. Algorithm

The whole Fast Multipole Method, can be divided into five major steps. Some of those steps can be executed concurrently but for the sake of simplicity, the sequential version shall be presented for now.

The original terminology from [GR87] only separates the Algorithm in an Upward- and Downward Pass. To stress the importance of the actual calculations of interactions between pseudo particles for this thesis, this part of the Downward Pass is here described as the Horizontal Pass.

For this explanation, a rectangular domain is assumed, which is structured in cells and has an arbitrary distribution of particles.

Upward Pass

The idea of the first phase is to propagate the information of particles up the tree of expansion, hence, it is called the Upward Pass. First and foremost the information of all particles need to be pooled to multipole expansions. This can either be done by an adaptive cluster-finding approach or simper by using a rigid grid of cells, which is assumed here. For every cell, a multipole expansion is created using the particle-to-multipole ($P2M$) operator.

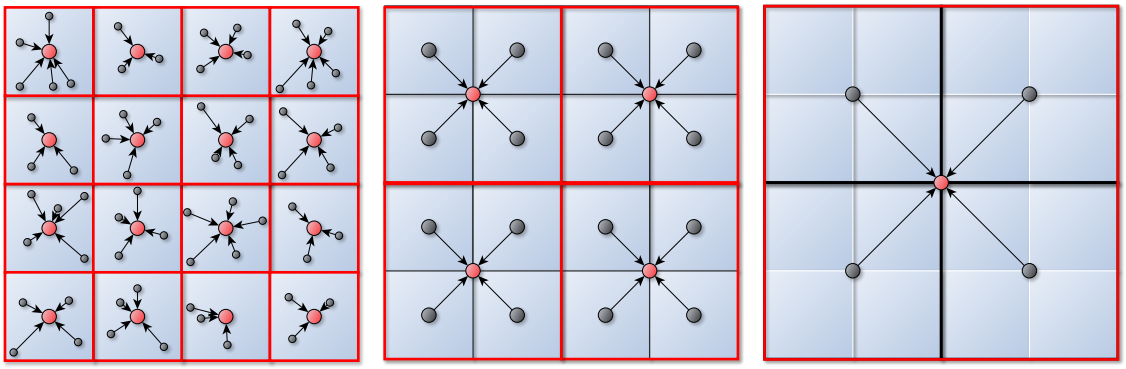


Figure 2.5.: Contributions of (pseudo-) particles to expansions during the Upward Pass. From left to right (pseudo-) particles (black) are hierarchically aggregated to pseudo particles (red) that represent a bigger part of the domain, starting with the real particles in the right. Red borders show the size of the cell the respective red pseudo-particle covers.

Subsequently, the following levels of the tree can be constructed. Every new multipole expansion combines 2^d, here four, expansions of the previous level in the pattern visualized in Figure 2.5. The creation of a multipole expansion from others is done by the means of the multipole-to-multipole (*M2M*) operator. This process is repeated recursively until only one multipole expansion representing the whole domain is constructed.

Horizontal Pass

With all multipole expansions updated, their influence on the local expansions can be calculated. This operation is done with the multipole-to-local (*M2L*) operator. Here, the multipole expansion is used to simulate the effect of the potential of a distant cluster on a local target, which is approximated by the local expansion. As described in Figure 2.4, this uses the optimal properties of both expansions. Since the multipole expansion is not a good approximation for short distances, only cell pairs that are separated by a Chebyshev distance greater than one are considered. Figure 2.6 colors these ignored cells in white. Cells with a Chebyshev distance greater one shall be called well-separated.

The first tree level to have enough cells to compute M2L operations contains four cells per dimension, as seen in the left part of Figure 2.6. This results for every cell on the next level in an area whose potentials have already been considered at the previous level. In the right part of Figure 2.6 this is depicted by the gray area.

The M2L operation can, for uniform grids, be significantly accelerated by calculating it in the Fourier space, which is described in detail in [Gal16]. For the Fast Multipole Method Algorithm, this has no further consequences except the transform of the source and target cell to and from Fourier space right before and after the M2L operation.

Downward Pass

After the Horizontal Pass, every local expansion contains the influence of all well-separated multipole expansions in a radius of Chebyshev distance of three on the same level. The goal

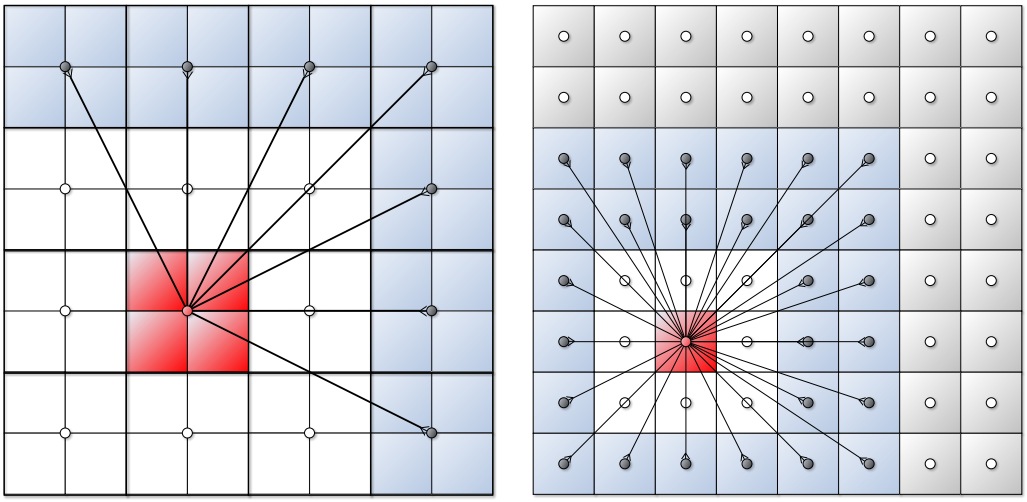


Figure 2.6.: Patterns for M2L operations. All cells who are not well-separated from the target cell and whose influence was not yet propagated to the target are included. Left: All well-separated multipole expansions (blue cells) are taken into account for the target local expansion (red). Direct neighbors (white) are ignored. Right: On the next level only the direct expansions within the neighbors from the level before need to be considered. Not well-separated neighbors are ignored again.

is for the local expansions on the leaf level to contain the full influence of all long-range (= well-separated) interactions. For this, the information of all local expansions that overlap over the levels need to be propagated to the leaf level using the local-to-local (*L2L*) operator. This process is visualized by the left and middle part of Figure 2.7. Every (pseudo-) particle represents its respective cell, which is confined by a line in the same color as the particle. Black source pseudo-particles propagate their information to the red target pseudo-particles. Traversing the tree, this scheme incorporates the influence of every well-separated multipole expansion on every local expansion.

Finally, since the first part of the Downward Pass aggregates all necessary information in the leaf cells of the expansion tree, the influences on the real particles can be calculated from their respective local expansions. For this, the local-to-particle (*L2P*) operator is used.

Near-Field Evaluation

The evaluation of all far-field interactions have left an area around every cell that is not yet accounted for, which corresponds to the white area seen in Figure 2.6 on the lowest level. This area shall be what is considered in the near-field Evaluation. This is essentially a direct evaluation of all pairwise particle interactions. The operator computing all interactions between two cells is called particle-to-particle (*P2P*).

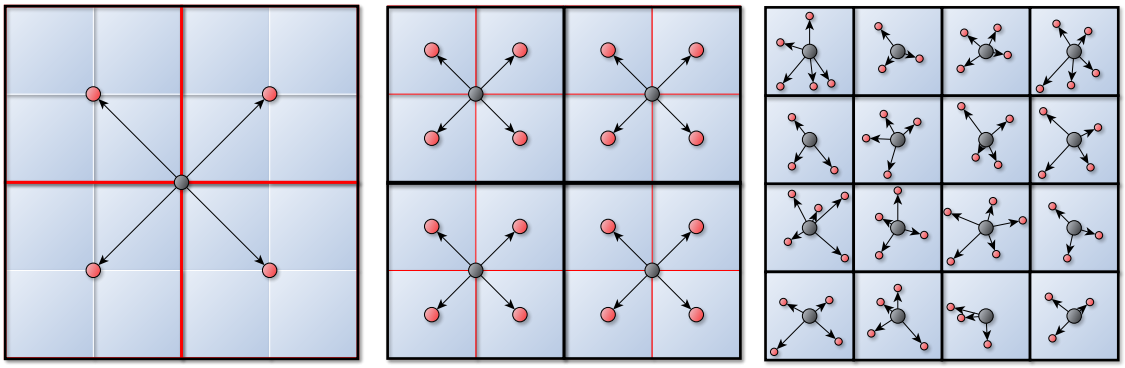


Figure 2.7.: Contributions of (pseudo-) particles to expansions during the Downward Pass. From left, representing the top of the tree, to right, representing the real particle level, (pseudo-) particles (red) are updated by local expansions (black).

Pseudo-Code Implementation

The detailed description from above can also be formulated as a pseudo code as seen in Algorithm 1.

2.3.3. Computational Complexity

As mentioned in the Introduction, the calculation of particle interactions in molecular dynamics simulation is an $N-$body problem. The direct computation of all pairs has a complexity of $O(N^2)$ where N is the number of particles in the system since every particle has to interact with every other.

An improvement to the computational complexity can be achieved by exploiting the idea of only calculating long-range interactions to pseudo-particles instead to all individual real particles. Clustering particles to pseudo-particles, for example by employing the multipole expansion, results in a tree of height $O(\log(K))$ where K is the number of cells in the domain on the last level.

Let s_d be the subdivision factor of the tree which is depending on the number of dimensions considered in the simulation. Every cell is divided into two cells per dimension:

$$s(d) = 2^d \tag{2.4}$$

For the three-dimensional case considered here $s_3 = 8$ which produces an octree. The number of nodes in such a tree is given by a geometric series:

Algorithm 1: Fast Multipole Method

Input: d dimensional Domain D containing particles P structured as cell grid

```
// Upward Pass
foreach cell do
    foreach particle in cell do
        Build leaf level multipole covering all particles in the cell using P2M
for level ← maxLevel to 0 do
    for m ← 0 to multipolesInLevel by 2^d do
        Build multipoles from 2^d adjacent multipoles using M2M
// Horizontal Pass
for level ← maxLevel to 0 do
    foreach Multipole m_t do
        foreach Multipole m_s ← radius_Chebyshev(m_t, m_s) ≤ 3
          and radius_Chebyshev(m_t, m_s) > 1 do
            Compute local expansion using M2L from m_s to m_t
// Downward Pass
for level ← 2 to maxLevel do
    foreach Multipole m_t do
        Add contribution of parent(m_t) local expansion to local expansion of m_t
          using L2L
// Far-Field
foreach cell do
    foreach particle in cell do
        Evaluate far-field influence of local expansion corresponding to this cell using
          L2P
// Near-Field
foreach cell c_t do
    foreach c_s ← neighbor cells and c_t do
        foreach particle p_t ← c_t do
            foreach particle p_s ← c_s do
                Calculate pair-wise interaction using P2P
```

$$\sum_{i=0}^{\log_{s(d)}(K)} s(d)^i = \frac{1 - s^{\log_{s(d)}(K)+1}}{1 - s(d)} \tag{2.5}$$

$$= \frac{1 - s(d) \cdot s(d)^{\log_s(K)}}{1 - s(d)} \tag{2.6}$$

$$= \frac{1 - s(d) \cdot K}{1 - s(d)} \tag{2.7}$$

As $s(d)$ is constant for a given scenario, Equation 2.7 lies in $O(K)$. Since K is typically chosen proportionally to N and typically $K \ll N$, it can safely be assumed that asymptotically $O(K) \in O(N)$.

Not every tree node can interact with every particle since expansions that include the particle or are not well-separated to it must be disregarded or included via near-field evaluations. However, if all single particles are targets like in the middle part of Figure 2.8, it is still necessary to traverse the whole height of the tree for every particle. Thereby this only reduces the complexity to $O(N \cdot \log(N))$. An example of an algorithm that is similar to this approach is the Barnes-Hut algorithm proposed by Josh Barnes and Piet Hut in 1986 [BH86].

Based on this, the advantage of the Fast Multipole Method is that it also clusters the targets for the long-range interactions by using the local expansion. The differences in the approaches are visualized in Figure 2.8. In this Figure, it can be seen that the interaction between two pseudo-particles can be calculated in constant time. However, also the creation of the pseudo-particles need to be accounted for. Greengard and Rokhlin show in their initial paper [GR87] that this can be done in $O(M)$ where M is the number of particles the pseudo-particle represents. This number is constant for all but the leaf level due to the tree structure. For the leaf level, it can also assumed to be constant, since the number of multipole expansions, like the number of cells K is typically chosen proportional to N. Therefore it can be concluded that $M \in \Theta(1)$.

Like K, M is typically proportional to N and $M \ll N$ which is why it can also be assumed that asymptotically $O(M) \in O(N)$.

Consequently, when factoring in that the *M2L*-operator consumes constant time and is for every target cell executed $6^d - 3^d$ times, the whole Horizontal Pass, line 7 in Algorithm 1 can be computed in $O(N)$.

The same complexity is true for the second part of the Upward Pass, starting in line 4, and the Downward pass in line 11, since they both traverse the whole tree of size $O(N)$.

However, since in the end, every particle needs to be accounted for and updated there need to be parts of the algorithm which traverse over the whole particle set and therefore have a complexity of $O(N)$. These are the first part of the Upward Pass in line 1 and the far-field evaluation from line 14. Also, when assuming to use a Linked-Cell algorithm for the near-field computation, the complexity of the last part of the algorithm, starting from line 17, lies in $O(N)$.

So to sum up, all loops in Algorithm 1 are in $O(N)$, therefore the whole algorithm for the Fast Multipole Method is in $O(N)$.

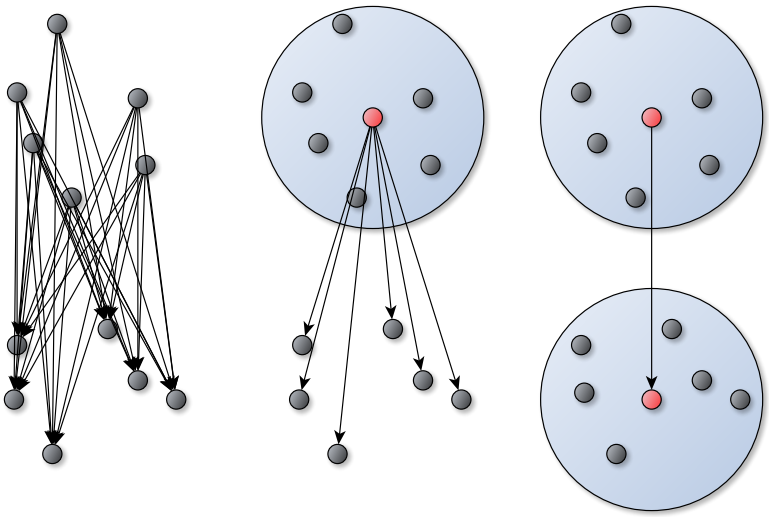


Figure 2.8.: Interaction techniques for long-range particle interactions.
Left: Naive pairwise interaction $O(N^2)$.
Middle: Clustering distant sources to one pseudo-particle (red) $O(N)$.
Right: Clustering both sources and targets to pseudo-particles (red) $O(1)$.

3. Description of Tools

This thesis was made using the already existing codes of the task based parallelization framework *QuickSched* and the molecular dynamics code *ls1-mardyn*. *QuickSched* was chosen because of its high flexibility and simplicity. The main target platform was a new *Xeon Phi* processor of the *Knights Landing* architecture. However, also *Ivy Bridge* processors were used to compare the impact of the architecture.

3.1. QuickSched

The library *QuickSched* is a tool for parallelizing code by the means of subdividing it into tasks.

It was developed by Pedro Gonnet[1,2], Aidan B.G. Chalk[1], and Matthieu Schaller[3]. They also work on *SPH With Inter-dependent Fine-grained Tasking (SWIFT)*[4], which is an open-source astrophysics simulation by the Institute for Computational Cosmology (ICC)[5] and the Institute of Advanced Research Computing (IARC)[6] at the University of Durham. The *QuickSched* library is the result of the back-port of a task based parallel approach for smoothed particle hydrodynamics in SWIFT [Gon15].

Written in *C*, *QuickSched* can be used with either *OpenMP* or *POSIX Threads (Pthreads)* as the underlying threading mechanism. For this thesis, *OpenMP* was used exclusively.

The distinguishing feature of *QuickSched* is its ability to model tasks that cannot be executed concurrently, but whose order of execution is irrelevant, by the means of lockable resources. This makes it more flexible than for example the explicit task mechanism offered by the current *OpenMP* version 4.5[7]. Furthermore, dependencies between tasks and organization of resources through hierarchies are supported. Since this thesis only contains a minimal introduction to *QuickSched*, a more detailed description of it and benchmark results can be found in [GCS16], or for the Knights Corner architecture in [Gra17].

In *QuickSched*, a `task` is a struct which holds, among other things, information on its dependencies and locks. The former which are the ids of the tasks it unlocks. The latter are the ids of the resources the task locks. Additionally, each task contains an integer indicating the type of the task, which is later used to determine what the task actually does. Also, every task is assigned a cost upon creation or by measuring the ticks it took to finish during the last execution.

[1] School of Engineering and Computing Sciences, Durham University, United Kingdom.
[2] Google Switzerland GmbH, Zürich, Switzerland.
[3] Institute for Computational Cosmology, Durham University, United Kingdom.
[4] `http://icc.dur.ac.uk/swift`
[5] `http://icc.dur.ac.uk`
[6] `https://www.dur.ac.uk/iarc`
[7] `http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf`

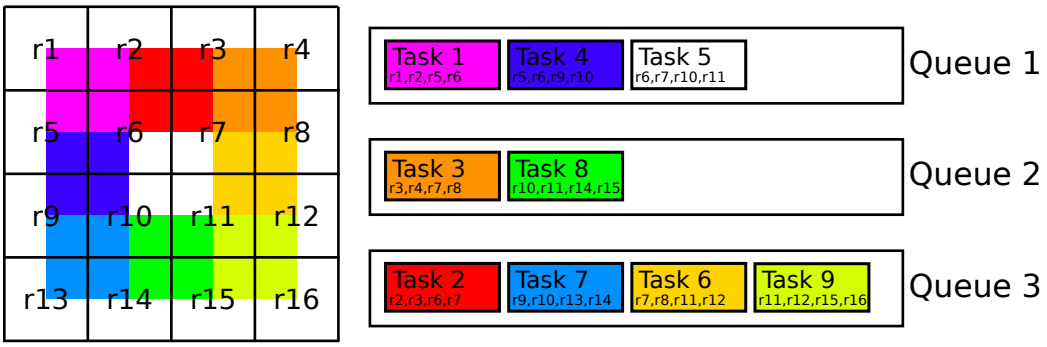


Figure 3.1.: Left: Domain divided into resources. Colors indicate tasks and which resources they require.
Right: Tasks distributed over queues.

For the actual scheduling process, a weight is calculated for each task. To find the critical path in an application the weight consists of the task's cost and the maximum weight of all the tasks it unlocks:

$$weight_i = cost_i + \max_{j \in \text{unlocks}_i} \{weight_j\} \tag{3.1}$$

The central component of the library is the `scheduler` struct. Whenever a task or resource is generated, its id and properties are saved there. It is used to resolve dependencies between tasks, order them according to the applications critical path, and distribute them over queues, which are processed by the different threads. A task is added to a queue when all its dependencies are resolved, but its resources do not necessarily have to be available at this moment.

Figure 3.1 sketches a possible situation as it can be encountered in the code this thesis considers. The white squares represent resources, for example, particle cells, and the colored squares tasks that calculate interactions between cells. Each task locks all resources it touches and evaluates these cell's interactions. This means the purple task one and blue task four are mutually exclusive since they both depend on the resources five and six. As no task depends on any other task, all dependencies are resolved and all tasks are distributed over the queues, which could result in the situation on the right of the Figure.

The distribution and ordering of the tasks inside the queues is a critical aspect since this determines the actual scheduling. *QuickSched* tries to put tasks which use the same resources in the same queues to benefit from caching. For the ordering, a so-called max-heap is used that compares tasks by their assigned cost. This means every k-th element's cost are larger than the $2k + 1$st and $2k + 2$nd. The benefit of this structure is a lower complexity of $O(\log n)$ for maintaining it as opposed to $O(n)$ for a strictly sorted queue, where n is the number of tasks in the queue. Though not providing optimal guarantees for the ordering of the tasks, according to [GCS16] it "[...] turns out to be sufficient in practice".

The desired number of queues has to be provided upon initialization of the `scheduler` while the number of threads used to process them has to be chosen upon execution of it.

In order to specify what the tasks actually do, the user has to define a `runner()` function

that chooses behavior according to the aforementioned task type. The required signature and a typical body is sketched in Listing 3.1.

```
1 void runner(int type, void *data){
2     switch(type)
3         case taskType1:
4             // do stuff using data
5         case taskType2:
6             // do other stuff using data
7 }
```

Listing 3.1: General form of a typical runner() function.

Finally, if the distribution of tasks over the queues was not sufficiently balanced and a thread finds its queue to be empty while there are still tasks to be executed, it tries to steal a task from a randomly selected other queue.

3.2. ls1-mardyn

The molecular dynamics code for which this shared memory parallelization was written is *large systems 1 - molecular dynamics (ls1-mardyn)* [8]. It is currently developed by the High Performance Computing Center Stuttgart (HLRS)[9] at the University of Stuttgart, the Laboratory for Engineering Thermodynamics (LTD)[10] at the University of Kaiserslautern, the chair for Scientific Computing in Computer Science (SCCS)[11] at Technische Universität München, the chair for Thermodynamics and Energy Technology (ThEt)[12] at the University of Paderborn, and the University of Darmstadt[13].

The source code is mainly written in *C++* and is publicly available as free software under a BSD license [NBB+14].

ls1-mardyn is able to model molecule interaction based on the Lennard-Jones and Coulomb potential to simulate van der Waals and electrostatic dipole and quadrupole forces. The code is specialized for huge numbers of multi-centered particles that can carry multiple charges. It was demonstrated that *ls1-mardyn* is highly scalable and is capable of simulating up to 4.125×10^{12} molecules [EHB+13] using Linked-Cells interaction mechanisms.

Included in *ls1-mardyn* is also an implementation of the Fast Multipole Method, which is optimized by a Fast Fourier Transform acceleration [Gal16]. There also exists a distributed memory MPI parallelization whose scalability has also already been demonstrated [Obe16]. At the moment, however, no comprehensive shared memory parallelization for *ls1-mardyn*'s Fast Multipole Method is available, which is the goal of this thesis.

3.3. Computation Platforms

The *QuickSched* parallelization was developed for arbitrary x86 platforms. However, experiments were mainly conducted on two platforms with different architectures, namely *Knights*

[8] http://www.ls1-mardyn.de
[9] http://www.hlrs.de/home
[10] http://thermo.mv.uni-kl.de
[11] https://www5.in.tum.de
[12] http://thet.uni-paderborn.de
[13] https://www.tu-darmstadt.de

Landing and *Ivy Bridge*, which are described below in order to be comparable.

3.3.1. CooLMUC3 / Knights Landing

The primary target platform for this thesis and on which most tests that are shown in Part III were conducted is the CooLMUC3 [14] omnipath-connected many-core cluster located at the LRZ[15]. The cluster consists of 32 nodes, each with one *Intel Xeon Phi 7210F* processor of the *Knights Landing* architecture. These processors consist of 64 x86 cores with a clock frequency of 1.3 GHz[16]. Each core contains two 512 Bit wide vector processing units (VPU) that implement the AVX-512 instruction set extension, allowing for a maximum of 32 double precision floating point operations per cycle when using fused multiply-add (FMA) operations[17]. They are also capable of so-called four-way Hyper-threading, which allows for the concurrent execution of four threads per core.

Another interesting feature is the so-called High-Bandwidth Memory (HBM), which is 16 GB of MCDRAM that is directly integrated into the processor and replaces the L3-cache. It can be set to different operation modes to act as cache-memory, addressable memory, or a 50/50 hybrid of the two modes. For the course of this thesis, cache mode was used exclusively.

All these features make the *Knights Landing* architecture a good target for highly parallel shared memory codes and therefore the logical target platform of this thesis.

On CooLMUC3, the *Intel C/C++ Compiler (ICC)* version 17.0.4 was used for compilation and it was linked against *Intel Math Kernel Library (MKL) 2017*[18] update 3 for a *Knights Landing* compatible implementation of the FFTW functions.

3.3.2. SuperMIC / Ivy Bridge Nodes

SuperMIC[19] is part of the SuperMUC[20] cluster at LRZ. It consists of 32 nodes each comprising of two *Intel Xeon E5-2650 v2 (Ivy Bridge)* host processors and two *Intel Xeon Phi 5110P (Knights Corner)* accelerator cards connected via PCIe 2.0. Since for this thesis only the *Ivy Bridge* host processors were used, no further description of the accelerators will be given here. For further details and *ls1-mardyn*'s performance using *QuickSched* for the computation of short-range forces on these processors, the interested reader is redirected to [Gra17]. The host processors were chosen to provide a reference of the performance of the code on a more classical architecture and see the impact of the architecture.

On SuperMIC, the *GNU Compiler Collection (GCC)* version 7.2.0 was used for compilation and the FFTW[21] library version 3.3.3.

[14] https://www.lrz.de/services/compute/linux-cluster/coolmuc3
[15] https://www.lrz.de
[16] https://ark.intel.com/products/94709/Intel-Xeon-Phi-Processor-7210F-16GB-1_30-GHz-64-core
[17] https://www.lrz.de/services/compute/linux-cluster/coolmuc3/overview
[18] https://software.intel.com/mkl
[19] https://www.lrz.de/services/compute/supermuc/supermic
[20] https://www.lrz.de/services/compute/supermuc/systemdescription
[21] http://www.fftw.org

Feature	CooLMUC3	SuperMIC (host Processors)	SuperMIC (Accelerators)
Nodes	148	32	32
Processors per Node	1	2	2
Cores per Processor	64	8	60
Threads per Node[a]	256	32	240
Vector Instruction Extension	AVX-512	AVX	KNC
Frequency [GHz]	1.3	2.6	1.05
L1 Cache [KB] per Processor	64×32 instr. 64×32 data	8×32 instr. 8×32 data	60×32 instr. 64×32 data
L2 Cache [KB] per Processor	32×1024	8×256	60×512
L3 Cache [MB] per Processor	-	20	-
Memory per Node [GB]	96 + 16 HBM	64	8 (Per Accelerator)
Memory Bandwidth [GB/s]	80.8 (460 for HBM)	59.7	320
Interconnect	Intel Omnipath	Mellanox Infiniband FDR14	PCIe 2.0 to host

Table 3.1.: Overview of the technical details of the used platforms. Data taken mainly from `https://www.lrz.de/services/compute/linux-cluster/coolmuc3/overview` for CooLMUC3 taken and from `https://www.lrz.de/services/compute/supermuc/supermic` for SuperMIC .

[a] Including Hyper-threading

4. Related Work

As already described in Section 2.3, it is difficult to compare specific implementations of the Fast Multipole Method. Nevertheless, it is always good practice and beneficial to see where similar attempts have been made and what conclusions were drawn, since task based approaches of the Fast Multipole Method have already been made.

Detailed comparisons of state-of-the-art methods for long-range interactions like multigrid-based-, P3M, Fast Fourier Transform based-, and the Fast Multipole Method were compared in [AFH+13]. It is concluded that the Fast Multipole Method, although not yet widely adopted, offers the most efficient performance and scalability. However, most of the time the complex choice of parameters for the simulation and selected method have a significant impact on accuracy and performance. Additionally, the authors compare the performance of the approaches on different architectures. They report that on platforms with low processor frequency the Fast Multipole Method outperforms all other tested algorithms.

Codes from different projects such as *ExaFMM*[1], which was originally developed by Rio Yokota and Lorena A. Barba at the Boston University[YB12], make of use the so-called dual tree traversal to determine what interactions need to be evaluated between which cells [Yok13]. This approach is combined with dynamically spawned tasks by the *MassiveThreads*[2] library [NT14] to provide a flexible and scalable parallelization[TNYM12].

In contrast, the here presented approach builds the whole task structure in advance by directly determining interaction partners through index calculations. Therefore, a large amount of tasks to be executed is directly available at the beginning of the computation and does not have to be generated at every time step.

The *StarPU*[3] library is another task programming library aimed for hybrid CPU / GPU platforms. Compared to *QuickSched*, it has more elaborate features, for example defining extensions to the *C* language through the *StarPU GCC* plug-in and then being used in a very similar manner like *OpenMP* through compiler pragmas. In [ABC+14], several algorithms are explored and compared using *OpenMP* and *StarPU*. It is concluded that *StarPU* offers more high parallel performance because fewer synchronization points are needed due to greater flexibility. The authors also highlight the strong impact of the choice of the scenario on the balance of the load within the Fast Multipole Method's steps and the thereby induced difficulties in load balancing.

Since the load balancing in *QuickSched* is done on an empirical basis by measuring the time each task took in the previous time step, a highly dynamic load balancing is to be expected. However, scalability can still be negatively impacted by a suboptimal parametrization of the Fast Multipole Method leading to an imbalance between *P2P*- and *M2L*-operations [Kab12].

A further related example is the so-called data-driven Fast Multipole Method implementa-

[1] `http://www.bu.edu/exafmm`
[2] `https://github.com/massivethreads/massivethreads`
[3] `http://starpu.gforge.inria.fr/`

tion described in [LY14]. It uses the *QUeuing And Runtime for Kernels (QUARK)*[4] runtime environment to asynchronously schedule tasks at runtime. *QUARK* aims to schedule tasks according to the data they use to achieve high re-usability. *QuickSched* also considers this by taking the resources used by a task into account when assigning it to queues.

The author of [LY14] also remarks that creating tasks for every single *M2L*-operation induces an undesirable scheduling overhead due to the massive number of tasks since every cell has 189 *M2L*-interactions in three-dimensional space. A similar comparison is made in this thesis in Section 6.4.

The author also recommends not to use two-way operations since they block both cells, as they are both source and target of an operation. In [ABC+14] it is suggested to create blocks for the *M2L*-operation. Both of these findings were incorporated in the design of a task pattern for the *M2L*-step described in Subsection 5.2.3.

[4] `http://icl.utk.edu/quark/`

Part II.

Towards a Task Based Implementation of the Fast Multipole Method

5. Implementation

As mentioned in Section 2.3 and Chapter 4, many different implementation details and paradigms for the Fast Multipole Method exist. The here presented task based parallelization of a sequential implementation is designed for maximal flexibility during scheduling of computation tasks. Therefore it will be explained how the code subdivides the domain, how tasks are formulated, and what dependencies exist between them.

5.1. Domain Segmentation

Like the Linked-Cell Algorithm, the Fast Multipole Method is based on a segmentation of the domain in cells. The size of these cells does not necessarily have to be uniform, but since *ls1-mardyn* currently only supports cells of uniform size this thesis considers this case exclusively. However, the parallelization approach presented here does not prohibit a generalization to an adaptive cell size.

The structuring of the domain as a whole is visualized in Figure 5.1.

ls1-mardyn segments a given cuboid domain in a three-dimensional grid of uniform cells. These cells are the basic structure of the Linked-Cell Algorithm. On top of this grid, an octree of so-called multipole-cells is constructed. These cells contain both the multipole and local expansion of the area of the domain belonging to the cell. For every Linked-Cell-cell there is one leaf cell in the multipole tree. The root cell residing at tree level zero represents the whole domain.

At the edges of the domain, a layer of halo cells of width one is created in the Linked-Cell grid. There are no multipole-cells for these halo cells due to periodicity.

5.2. Tasks and Dependencies

The Fast Multipole Method consists of several operators described in Subsection 2.3.2, which are all employed in separate loops as shown in Algorithm 1. Therefore, a straightforward approach is to define a task type for each operator and then formulating the dependencies between them. It is important to consider that all tasks need to be initialized before they are executed. This means that they cannot depend on data or pointers generated during runtime since also their payload must be initialized in advanced.

For example, if a task needs the potential of a multipole expansion of a cell the payload cannot contain this potential since it is not known in advance. It must contain for example the cell id since this information can be computed in beforehand. From there the task needs to have enough privileges to be able to access the relevant information.

A summary of the number of tasks, dependencies, and locks with respect to the number of cells in the Linked-Cells data structure is presented in Table 5.1.

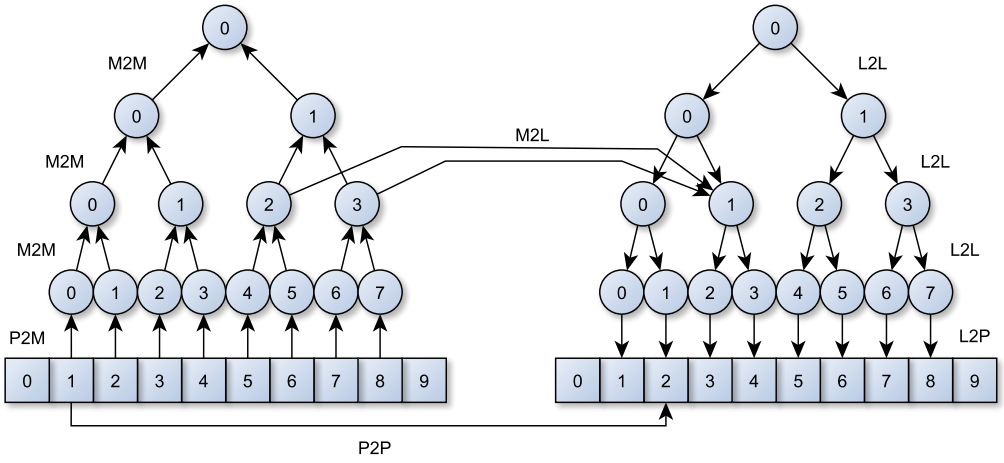


Figure 5.1.: Representation of the data structure and interactions of the Fast Multipole Method in *ls1-mardyn*. The left and right tree represent the exact same domain during upward and downward pass, respectively. Almost all *P2P*- and *M2L*-interactions are omitted for visibility.

5.2.1. P2M

The first part of the Upward Pass is an interaction between the Linked-Cell structure for the near-field computation and the last level of the multipole cell tree. In Figure 5.1 the former are depicted as squares and the multipole cells as circles.

One *P2M*-task executes exactly one *P2M*-operation between one source cell and its respective target.

Since the halo region of the Linked-Cell structure has no counterpart in the multipole cell tree no *P2M*-operations are executed there. This also means that the source and target cell differ in their index by the width of the halo in every dimension.

As there is no overlapping in either sources or targets there are no dependencies and all of these tasks can be executed in parallel.

5.2.2. M2M

The second part of the Upward Pass consists of interactions between the layers of the multipole cell tree. As seen in Figure 5.1, 2^d cells are contributing one cell in the next layer.

Here, to reduce the total number of tasks and dependencies, one *M2M*-task consists of all *M2M*-operations that share the same target. Thus, all *M2M*-tasks on one level are independent.

Since the idea of the Upward Pass is to propagate the information from the lowest level to the top, each *M2M*-task needs to wait until all of its source cells' *M2M*-tasks are complete. The exception are the *M2M*-tasks between the lowest and second lowest level where they need to wait for their respective *P2M*-tasks instead. This way, every *M2M*-task depends on 2^d other tasks in order to be unlocked.

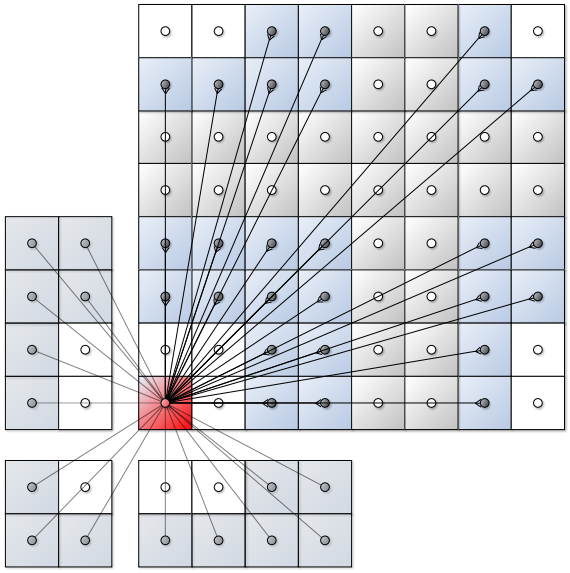


Figure 5.2.: Model of periodic boundary conditions for the *M2L*-step. Since the fainter cells in the separated blocks are out of the domain, their counterparts at the opposing side of the domain are used instead.

5.2.3. M2L

Like the term Horizontal Pass suggests, all *M2L*-operations are confined to their level. All *M2L*-tasks on different levels are independent. Also, all *M2L*-operations that do not share the same target but are on the same level can be executed in parallel.

It was decided that the Fast Multipole implementation for *ls1-mardyn* should simulate periodic boundary conditions [Kab12]. This has the advantage of avoiding effects at the edge of the domain or finite-system effects [Eck14]. Figure 5.2 shows how periodic boundary conditions are modeled for the extreme case of the cell in the lower left corner in a two-dimensional scenario. The red cell should interact with the fainter cells following the stencil described in Section 2.3.2. However, since the domain ends to the left and below with the red cell, the respective cells at the opposing sides of the domain are chosen. This corresponds to taking the coordinates of the target cells modulo the width of the domain on this level.

Since the *M2L* step is one of the most work-intensive steps of the Fast Multipole Method [KP06] it is of increased interest when aiming to optimize the algorithm. Therefore, different parallelization patterns were tested.

Pairs2Way The first pattern creates one task for the two directed *M2L*-operations between one pair of cells. Tasks which share a common cell cannot be executed in parallel since every cell in a task is used as a target.

As explained in Section 2.3.2, the *M2L*-operator can be accelerated by shifting the computation to the Fourier space. Therefore, every source multipole expansion and every target local expansion needs to be transformed before the actual *M2L*-operation can be performed. Since multiple *M2L*-tasks in this pattern have the same source or target these initializations cannot be part of the calculation tasks but need to be separate tasks that

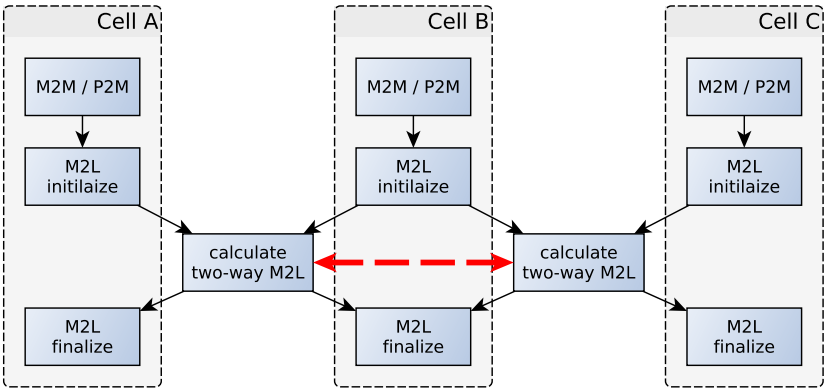


Figure 5.3.: Flowchart visualizing dependencies of *M2L*-tasks in the Pairs2Way task pattern. Black arrows represent a "depends on", red arrows a "cannot be executed simultaneously" relation.

precede or follow their respective calculation tasks. Consequently, a *M2L initialization* and a *M2L finish* task type is introduced for forward and back transform. Both task types transform the multipole expansion, as well as the local expansion of one cell. The task dependencies for three cells are described in Figure 5.3.

The advantage here is a very fine granularity for optimal dynamic load balancing. Also, this pattern can make use of the two-way *M2L* optimization explained in [Gal16].

The disadvantage, however, is a drastic increase in the number of tasks and dependencies compared to the next approach. This leads to a significant scheduling overhead, which is shown in Part III.

CompleteCell Similar to the grouping of *M2M*-operations, here one *M2L*-tasks consists of all *M2L*-operations for the same target cell. Following the algorithm described in Subsection 2.3.2, there are $6^d - 3^d$ *M2L*-operations per cell, which results for the three-dimensional case here in 189 sources per target. Both images in Figure 2.6 depict exactly one *M2L*-task by this pattern in the two-dimensional case. This also means that all these tasks can be executed independently as none of them share a target.

Since every local expansion is only used in exactly one task, the initialization and finalization of them can also be done in the *M2L*-calculation task to reduce the number of dependencies. The initialization of sources, however, still needs to be a single task per cell since sources are shared over multiple targets. Including the initialization of sources in the calculation task would mean that every source is initialized once per target and possibly concurrent which might lead to race conditions.

The *M2M*-task which targets a certain cell does not necessarily have to precede the *M2L*-calculation task targeting the same cell. Yet, as the two tasks share the same target they must not be executed at the same time. In order to model this kind of dependency with *QuickSched*, resources for the cell can be defined which are locked by either tasks. This whole pattern for the *M2L*- dependencies is depicted in Figure 5.4.

Although the two-way optimization cannot be used here, a higher degree of parallelism

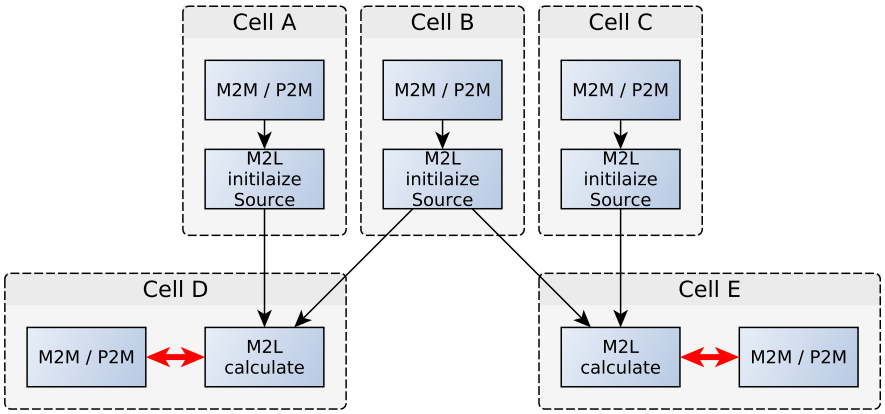


Figure 5.4.: Flowchart visualizing dependencies of *M2L*-tasks in the CompleteTarget task pattern. *M2M*-operations have the cell they are in as target.
Black arrows represent a "depends on", red arrows a "cannot be executed in parallel" relation.

can be reached since all calculation tasks are independent. This tradeoff is examined in [LY14] and the authors concluded that for dynamic task based scheduling a higher degree of parallelism is more advantageous which is supported by the experiments in Part III.

5.2.4. L2L

As seen in Figure 5.1, the *L2L*-step is very similar to the *M2M*-step except that it is a Downward instead of an Upward Pass. Similar to its equivalent one *L2L*-task consists of all *L2L*-operations that share the same source. Therefore, all *L2L*-tasks on the same level are independent.

Also similar to the Upward Pass, the idea here is to propagate all information to the lowest level. This means that in order to start a *L2L*-task, all tasks which convey information to this respective cell need to be completed. These tasks are the *M2L*-task that includes the finalization of the *M2L*-step for this target cell and the *L2L*-operation that targets it.

Since a *L2L*-task obviously has a local expansion as a target, it cannot be executed in parallel with the *M2L*-tasks that have the same cell as their target. Thus, the same *QuickSched* resource needs to be locked.

5.2.5. L2P

The last part of the Downward Pass works analogously to the *L2L*-step except that it targets particle cells of the Linked-Cell structure instead of local expansions.

Every *L2P*-task too has to wait for the *L2L*- and *M2L*-tasks that target its source cell.

As *L2P*-tasks are not the only tasks to target particle cells, there need to be *QuickSched* locks for each particle cell too, which are taken by these tasks. Each *L2P*-task locks 2^d particle cells.

5.2.6. P2P

The last step of the Fast Multipole Method, the near-field evaluation, works only on the Linked-Cell structure. Therefore, the P2P step is almost the same as the Linked-Cell algorithm described in Section 2.2, which means that every cell has to interact with the cells shown in Figure 5.5 and itself or the left part of Figure 5.7.

The first difference to the Linked-Cell algorithm is that there is no spherical cutoff radius for each molecule. For each, molecule all molecules in the surrounding cells are considered.

The second difference is that for every cell the relevant data has to be converted to a so-called Structure of Arrays (SoA) before it is used in any *P2P*-operation and after all *P2P*-operations involving the cell are done the data has to be migrated to the single molecules. This is done to facilitate vectorization and improve cache efficiency. Details about the Structure of Arrays approach in *ls1-mardyn* are described in [Eck14].

Structuring the tasks for the computations of the interactions in the *P2P*-step has to be done in the same way as described in [Gra17]. There, a task based version of a parallelization strategy that employs coloring is presented.

One task takes a cube of two cells per dimension and calculates all interactions involving the cell in the lower left front corner. Here, an optimization is employed by exploiting Newton's third law of motion. It states that for every directed force F enacted from a body A to a body B there is a force of the same magnitude in the opposite direction:

$$F_{A,B} = -F_{B,A} \tag{5.1}$$

Using Equation 5.1 it is possible to reduce the number of computations done in each *P2P*-operation by a factor $\frac{1}{2}$ by only calculating every interaction in one direction and also applying the resulting force with the opposing sine in the other direction.

When applying this pattern to every cell, all horizontal, vertical, and diagonally right upwards interactions are covered. In the example illustrated in Figure 5.5 starting from cell 6 the interactions with the cells 2,3,5,7,9, and 10 are covered by this. To also include the interactions on the other diagonal it suffices to evaluate the diagonal interaction in the two-by-two cell task block as seen in Figure 5.6, since the pattern is employed on every cell. In the three-dimensional case, there are more additional diagonals that need to be accounted for, as can be seen in Figure 5.7.

With every cell in a task being used as source and target, for every computation, all of one task's cells need to be locked by it and in order to start a task, the locks of all cells need to be available. For example, in Figure 5.6 task two, which covers the cells 2, 3, 6 and 7, is blocked by both, tasks one and two. Additionally, every *P2P*-task blocks 2^d *L2P*-tasks, since they too target these particle cells. Therefore, these small versions of tasks seem better suited for this algorithm than the adaptive larger versions suggested in [Gra17] as they block significantly fewer tasks.

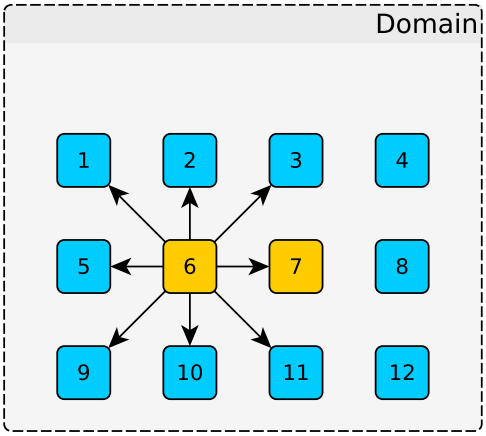


Figure 5.5.: Naive interaction pattern of one cell during the *P2P*-step in the two-dimensional case.
Source: [Gra17]

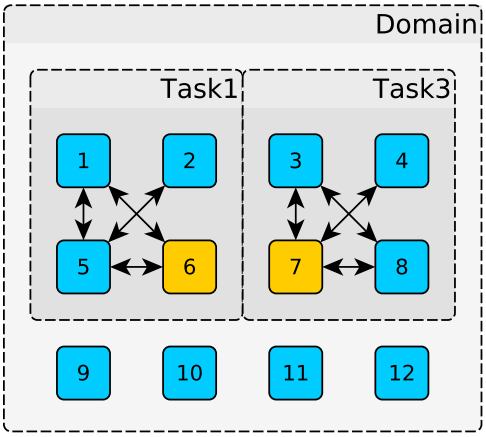


Figure 5.6.: Compact task pattern including Newton's third law of motion.
Source: [Gra17]

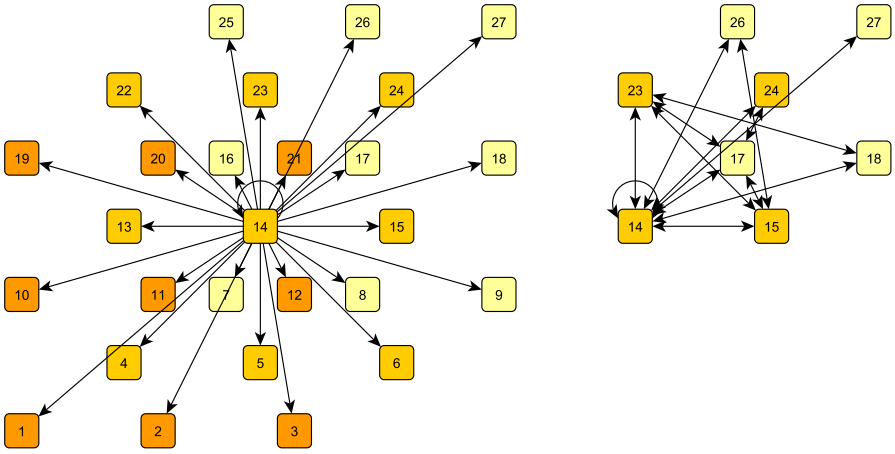


Figure 5.7.: Left: Naive interaction pattern of one cell in tree dimensions.
Rights: Compact interaction pattern.
Source: [Gra17]

5.2.7. Task Weighting

For all task types and approaches *QuickSched*'s automatic task weighting mechanism is used. It works by counting the ticks every single task takes to finish. This count is then used as the cost of the task for the next run of the scheduler. That also means that in the first run, which is the first time-step of the simulation, no task weighting is available which can result in suboptimal scheduling.

The method of counting the ticks is dependent on the platform for which *QuickSched* was compiled. A wide range of architectures, operating systems, and compilers are supported.

Tasktype	Number of tasks	Depends on	Unlocks	Blocks/Conflicts
P2M	K	-	$\frac{1}{8}$ *M2M* $\frac{1}{8}$ *M2L init*	-
M2M	$\sum_{l=1}^{\log_8(K)} 8^l$	8 *P2M* OR 8 *M2M*	$\frac{1}{8}$ *M2M* $\frac{1}{8}$ *M2L*	-
M2L init	$\sum_{l=1}^{\log_8(K)} 8^l$	8 *P2M* OR 8 *M2M*	$\frac{1}{189}$ *M2L*	1 *L2L* OR 1 *L2P*
M2L	$\sum_{l=1}^{\log_8(K)} 8^l$	189 *M2L init*	1 *L2L* OR 1 *L2P*	1 *L2L* OR 1 *L2P*
L2L	$\sum_{l=1}^{\log_8(K)} 8^l$	1 *M2L* 1 *L2L*	8 *L2L* OR 8 *L2P*	1 *M2L*
L2P	K	1 *M2L* 1 *L2L*	-	$8 \times \frac{1}{8}$ *P2P*
P2P init	$(\sqrt[3]{K}+2)^3$	-	*P2P*	-
P2P	$(\sqrt[3]{K}+1)^3$	*P2P init*	*P2P fin*	$8 \times \frac{1}{8}$ *P2P* $8 \times \frac{1}{8}$ *L2P*
P2P fin	$(\sqrt[3]{K}+2)^3$	*P2P*	-	-

Table 5.1.: Overview of all tasks and their dependencies relative to the number of Particle cells in the Linked-Cell structure K not including halo cells. The last three columns are the impact of one single task of this category. Here the "CompleteCell" *M2L*-task pattern is chosen.

For a full list, it is advisable to have a look at the definitions in the source code in `cycle.h`.

On x86-64 architectures when using the *GNU Compiler Collection (GCC)* or the *Intel C/C++ Compiler (ICC)* the `rdtsc`[1] assembler instruction is used.

Since the exact time of execution per task is dependent on many factors which cannot necessarily be influenced by the user, this task weighting is not fully deterministic. Thereby, this leads to a nondeterministic task scheduling.

[1] `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html`

Part III.

Verification and Validation

6. Experiments

In order to test the aforementioned implementations, the scenario depicted in Figure 6.1 was used. Here, a cube is randomly filled with uniformly distributed particles. In the middle of the domain, a sphere is placed, which has a radius of $\frac{1}{8}$ of the length of the domain. In this sphere, the density of particles is about 3.77 times higher and they are placed in a grid-like pattern. A part of this sphere can be seen in Figure 6.2, which is a slice of the center of the domain.

All particles are of the same type. They have one Lennard-Jones center and a σ and ϵ of one. Each molecule carries two charges of $+1$ and -1, respectively, offset from the center by 0.005 on the z axis in either direction.

Two sizes of the experiment are considered. The first with a side of length 80, which leads to a Linked-Cell structure of $16 \times 16 \times 16$ cells without halo cells containing 85805 molecules in total. The second version is of double width in each dimension, which leads to $32 \times 32 \times 32$ cells with a total of 675394 molecules.

This in-homogeneous setup guarantees a moderate imbalance in the computation time for tasks of the same type.

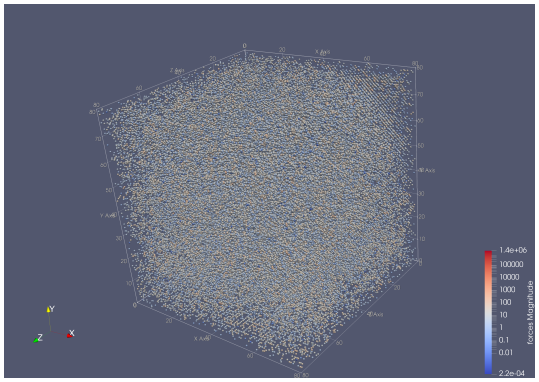


Figure 6.1.: Visualization of the full simulation domain.

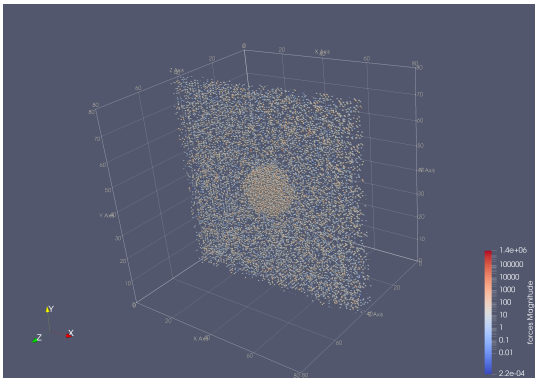


Figure 6.2.: Slice showing the denser center region.

6.1. Strong Scaling

Using the scenario just described, strong scaling tests were performed and different parameter configurations were tested. This was done mainly on the *Xeon Phi* processors described in Subsection 3.3.1 and also on the *Ivy Bridge* processors of Subsection 3.3.2 to highlight the effects of the architectures on the code.

Size is the length of the domain not measured in cells but in a unit length of about 5.29177×10^{-11} defined in [NBB+14]. Here, if not otherwise specified, a size of 80 is the default for all experiments.

The subdivision factor is a relative information on the number of cells the domain is subdivided in, whereby a subdivision factor of one corresponds to the number of cells described above in Chapter 6. A factor of two results in a split of every particle cell in eight equally sized cells. Consequently, the tree of multipole cells increases in size by exactly one level. This means more of the force calculation is moved to the long-range part.

Order stands for the order of the multipole and local expansion. A higher order typically brings the benefit of increased accuracy of the simulation, however, at the cost of an increased runtime.

The first Figure 6.3 compares the two task patterns that are described in Subsection 5.2.3. Here, in contrast to all other Figures, only time for the actual calculation steps, which are the *M2L*- and *P2P*-step, is taken into account. This is especially visible at the given time to solution. For all remaining tests and Figures, the "Complete Cell" approach was used.

Figure 6.4, Figure 6.5, and Figure 6.6 show the effects of different domain sizes, subdivision factors, and orders, respectively. Figure 6.7 and Figure 6.8 show combinations of different orders and subdivision factors on the *Xeon Phi* and *Ivy Bridge* platform.

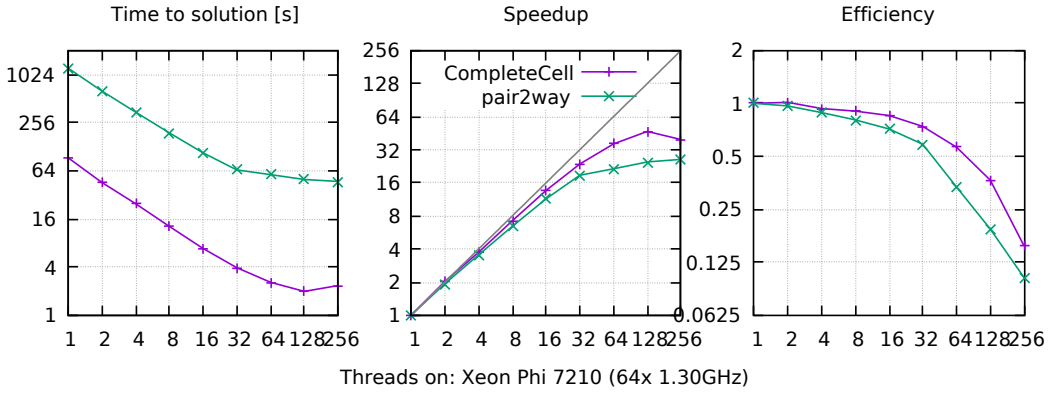


Figure 6.3.: Comparison of the two parallelization strategies of the *M2L*-step described in Subsection 5.2.3. Time to solution is here significantly lower as only the time for the *M2L*- and *P2P*-steps are taken into account.

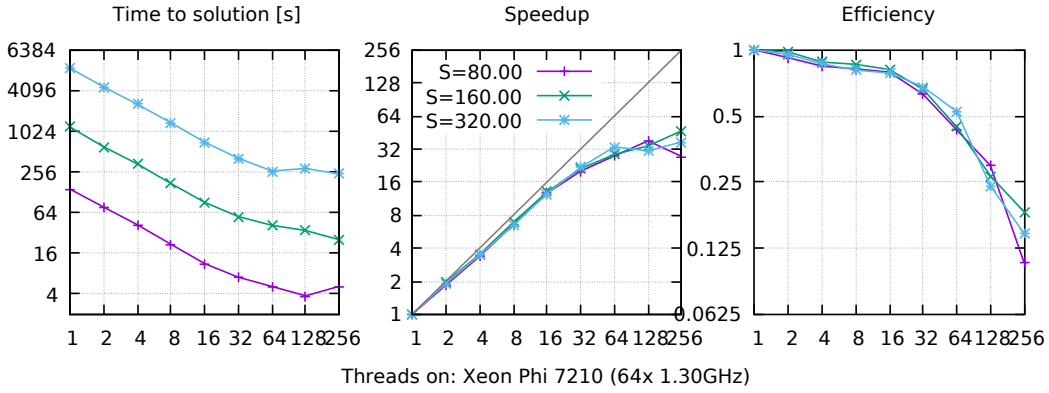


Figure 6.4.: Comparison of different domain sizes with order 10, subdivision factor 1, time steps 30.

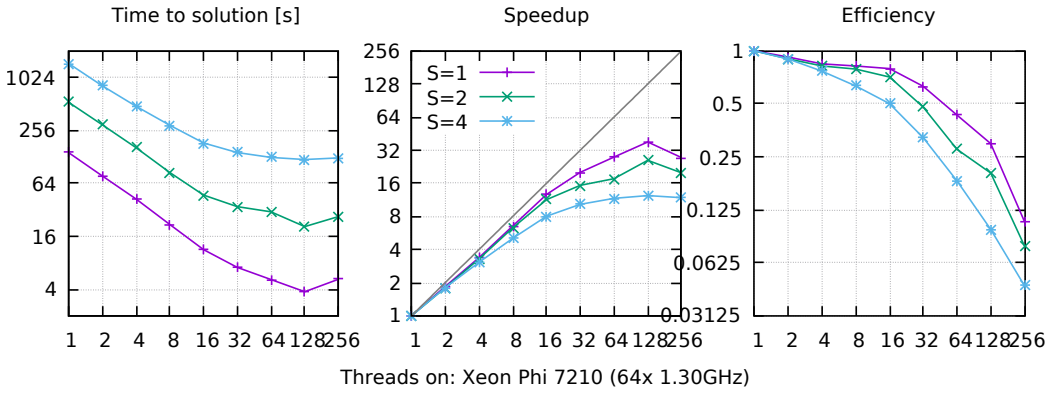


Figure 6.5.: Comparison of different subdivision factors with order 10, size 80, time steps 30.

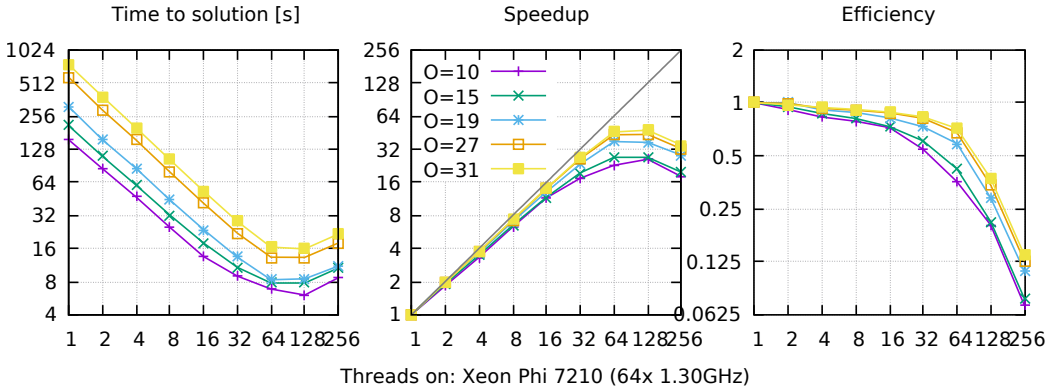


Figure 6.6.: Comparison of different orders with subdivision factor 1, size 80, time steps 30.

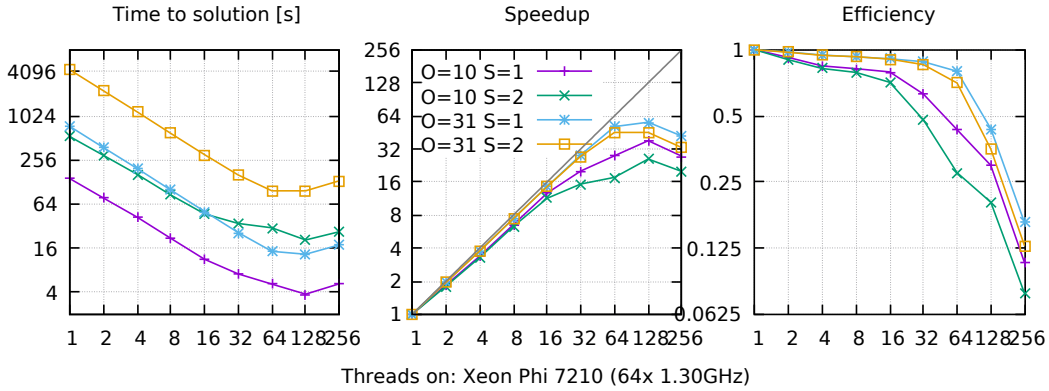


Figure 6.7.: Xeon Phi: Comparison of different orders and subdivision factors, size 80, time steps 30.

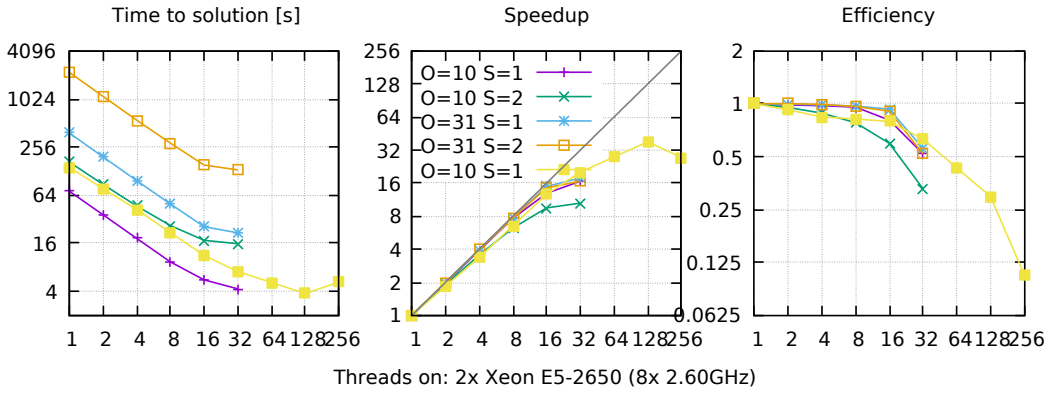


Figure 6.8.: Ivy Bridge: Comparison of different orders and subdivision factors, size 80, time steps 30. The yellow line is from the *Xeon Phi* platform for comparison.

6.2. Weak Scaling

Due to the limitation of *ls1-mardyn* to only allow for cubic sizes and with the edge length being multiples of 10, the weak scaling has to always increase by a factor of $2^3 = 8$ instead of 2^1. Since the available *Xeon Phi* processor has only 64 cores just three measurement points are possible when there should be a maximum of one execution thread per hardware thread, even when taking the four-way Hyper-threading into account.

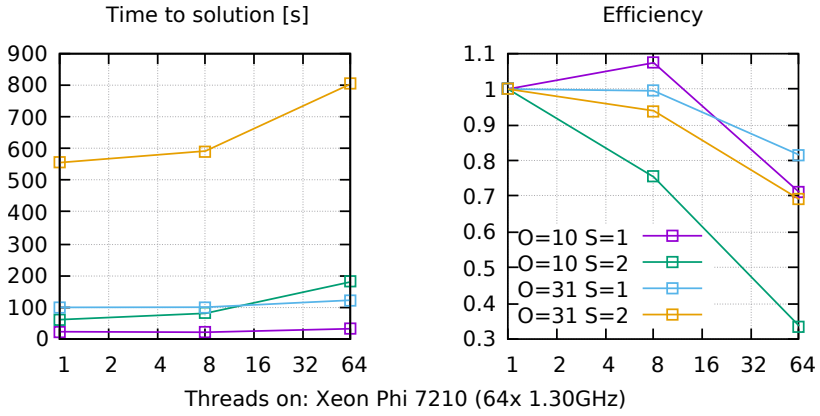


Figure 6.9.: Weak scaling: Size 40, 80, 160, Timesteps 30.

6.3. Task Timings

In this section detailed visualizations of the activity of the threads of the processor are shown. All Figures show a measurement on the *Xeon Phi* platform except Figure 6.22 and Figure 6.23, which are from the Ivy Bridge platform.

To achieve a high resolution, the `rdtsc`[1] assembler instruction was used before and after every task. This is the information represented on the x-axis.

[1] `https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html`

On the left-hand side always five time steps are shown. The right-hand side is always a closer lookup of the fourth time step, which is circled in red on the left. Colors indicate the task type as given in the keys.

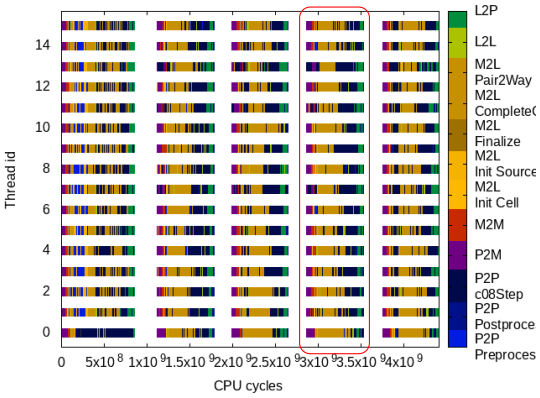


Figure 6.10.: Order: 10, Subdivision 1.

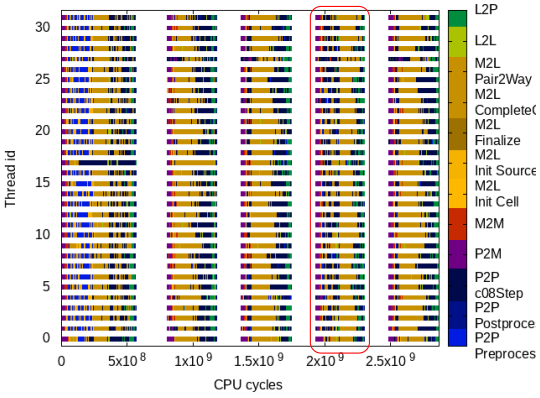


Figure 6.11.: Order: 10, Subdivision 1.

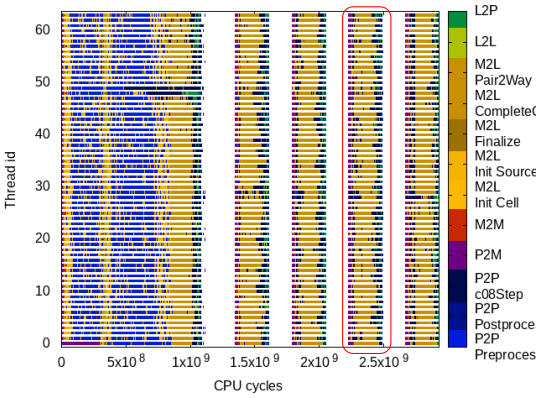


Figure 6.12.: Order: 10, Subdivision 1.

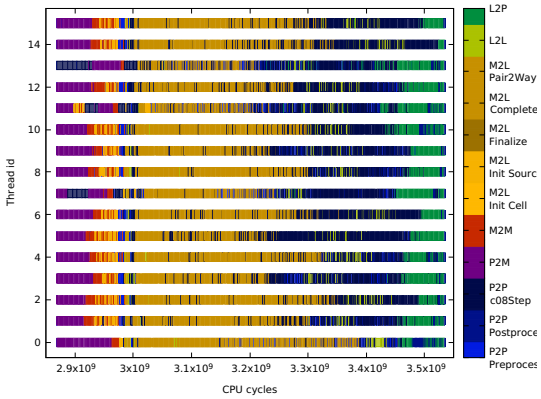


Figure 6.13.: Order: 10, Subdivision 1.

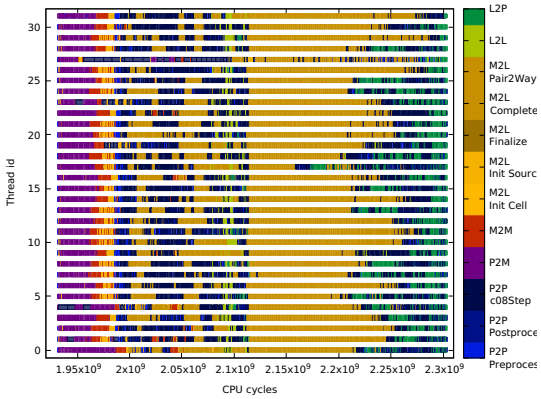


Figure 6.14.: Order: 10, Subdivision 1.

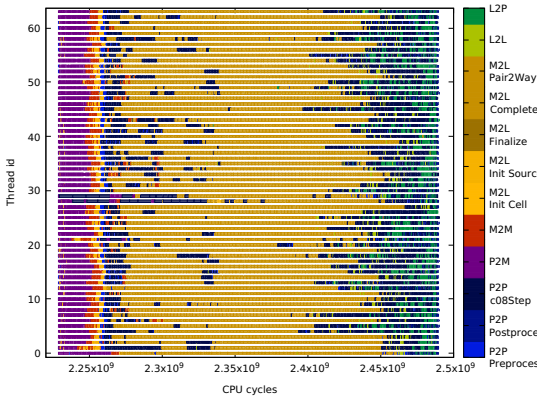


Figure 6.15.: Order: 10, Subdivision 1.

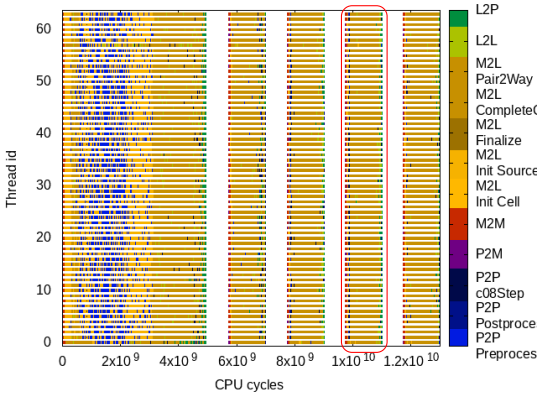


Figure 6.16.: Order: 10, Subdivision 2.

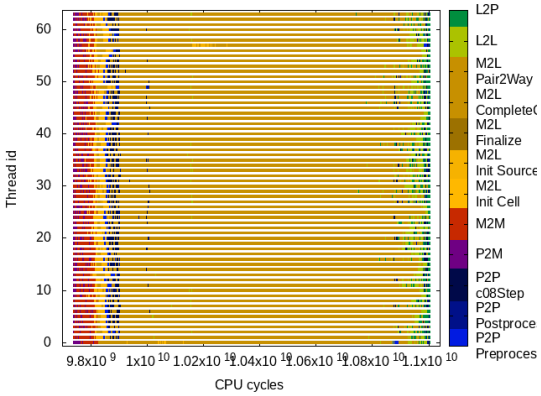


Figure 6.19.: Order: 10, Subdivision 2.

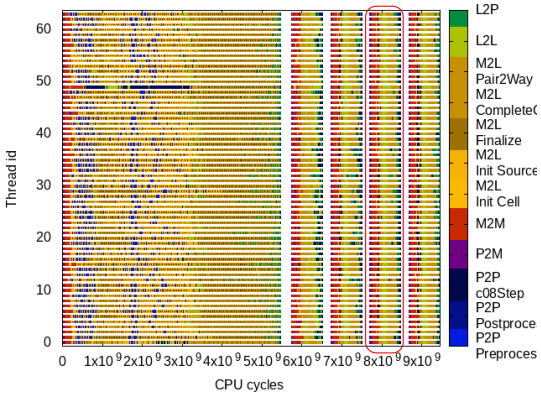


Figure 6.17.: Order: 31, Subdivision 1.

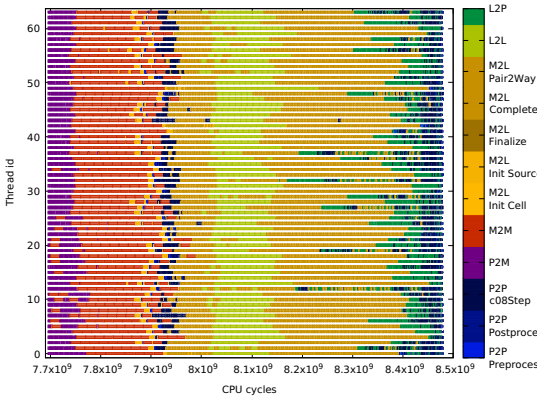


Figure 6.20.: Order: 31, Subdivision 1.

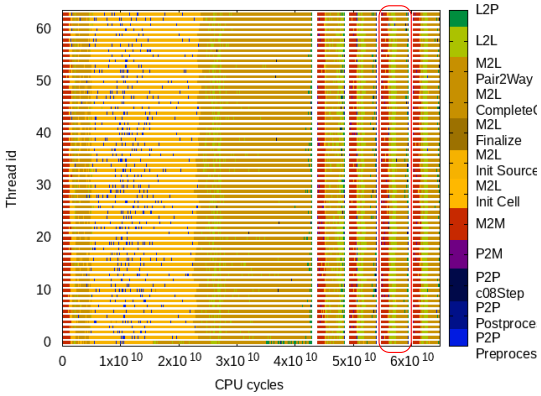


Figure 6.18.: Order: 31, Subdivision 2.

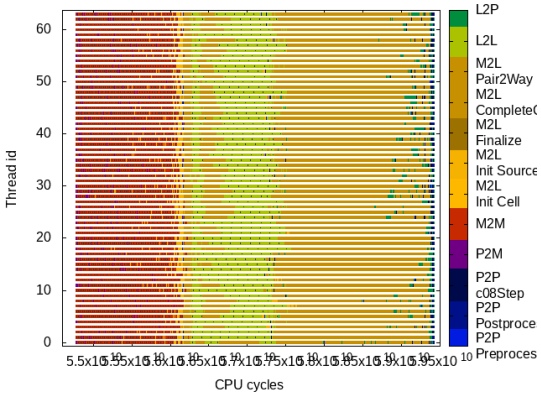


Figure 6.21.: Order: 31, Subdivision 2.

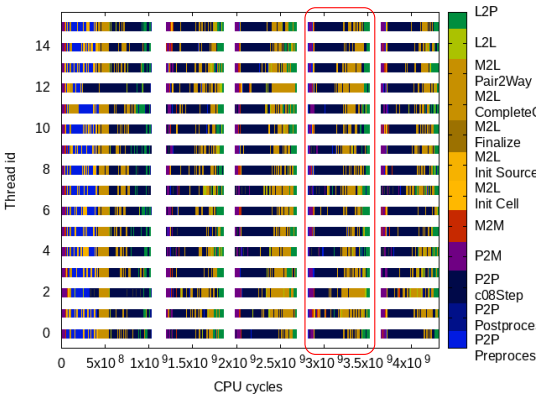


Figure 6.22.: *Ivy Bridge*: Order: 10, Subdivision 1.

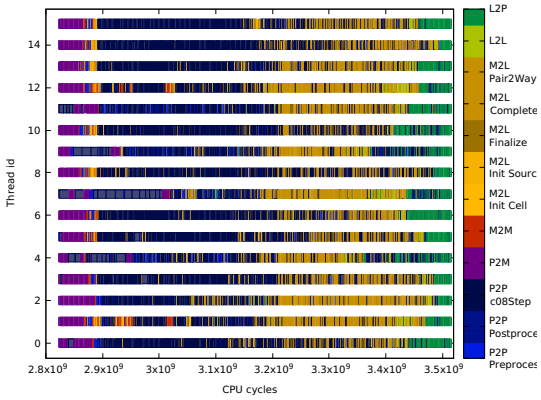


Figure 6.23.: *Ivy Bridge*: Order: 10, Subdivision 1.

6.4. Analysis

In High-Performance Computing (HPC) the typical goal is not only to "get the answer as fast as possible" but to use the available hardware as efficiently as possible. This is measured in parallel efficiency, which is the achieved speedup divided by the number of threads used.

Given the plots and figures from Chapter 6, an analysis of various parameters and their combinations shall be performed.

6.4.1. M2L-Task Patterns

The first aspect to analyze is the effect of the two task patterns for the *M2L*-step, which were described in Subsection 5.2.3. Looking at Figure 6.3 the "Complete Cell" approach is significantly faster and sustains an equal or higher parallel efficiency for all thread counts. This was to be expected since this is comparable to task patterns compared to [Gra17]. There it is also observed that it is beneficial to cluster operations that share resources to one task instead of creating a significantly higher number of tasks with many locks and dependencies between them since this notably increases the scheduling overhead.

For this reason, it was decided only to continue experimenting with the "Complete Cell" approach, so no further data on the "Pair2Way" approach was collected.

6.4.2. Domain Size

Comparing the performance using different domain sizes Figure 6.4 suggests an equally good scaling relative to each size. Only at size 80 with 256, there is a larger deviation. This suggests that for this number of threads the experiment is not large enough. However, running 256 threads already uses the four-way Hyper-threading and is therefore for now not of primary concern.

6.4.3. Subdivision Factor

Increasing the subdivision factor, as demonstrated in Figure 6.5, has a huge negative impact on the time to solution, which is expected, but also on the speedup and parallel efficiency. Adding one layer to the multipole cell octree roughly increases the number of cells by a factor of eight. At the same time, the total number of molecules stays the same so the actual computational expense of the average task decreases. This means the fraction of the total execution time of the scheduling increases. Reducing the size of the particle cells means that the cutoff that decides where the near-field computation ends and particles are considered to be in the far-field is narrowed. For the Fast Multipole Method, the effect is a shift of computational effort from the *P2P*- to the *M2L*-step. Also, the far-field part grows faster than the near-field part. This is because of the tree, responsible for computing the far-field, gains another layer but still has to compute all other layers. The near-field part just adopts the new size and has to cover the same number of cells as the last layer of the multipole cell tree. Therefore, also the other task types that operate inside the tree grow slightly more in computational expense than the *P2P*-part. All this can indeed be observed when comparing the task timing plots in Figure 6.15 and Figure 6.19, which are showing a single time step with the subdivision factors one and two, respectively.

6.4.4. Expansion Order

Increasing the order of the multipole and local expansion also has a negative effect on the time to solution. However, the effect on scaling and parallel efficiency is positive as can be seen in Figure 6.6. A higher order increases the computational expense of all operations that interact with a local or multipole expansion, which are all task types except those for the *P2P*-step. This leads to the exact opposite effect induced by the subdivision factor, which was described above. Here, the fraction of the total time of execution that represents scheduling decreases because of the fraction of computation increases. Comparing Figure 6.15 and Figure 6.20, it can be seen that the latter contains relatively fewer blue areas which indicates a decline in the dominance of the *P2P*-step. It can also be seen that from all task types that interact with either multipole or local expansions, especially the *L2L*- and *M2M*-tasks increase in dominance. Interestingly, they seem to grow in computational expense even faster than the dominant *M2L*-step. This can be led back to the optimizations through the Fast Fourier transform and order reduction techniques described in [Gal16].

From a user's perspective, according to Figure 6.6, an order of 19 seems to be a good compromise between increased runtime and parallel efficiency.

6.4.5. Combining Subdivision and Order Effects

In Figure 6.7 a comparison is made between combinations of high and low order and two subdivision factors. The blue and green line being very close in the left panel indicates that a drastic increase in the expansion order increases the total runtime roughly in the same way as raising the subdivision factor by one with the high order version being slightly slower. However, the higher subdivision factor does not scale equally well, which leads to the high order being faster starting from 32 threads. This is also a number of threads where the version with the higher subdivision factor slips under 50% parallel efficiency.

When combining both the high order and higher subdivision factor a strong increase in the time to solution can be observed. Nevertheless, the positive effects on the scaling and efficiency of the higher order clearly dominate the negative ones of the increased subdivision factor as the yellow line shows good efficiency for up to 64 threads.

Looking at the task timings in Figure 6.21 this is supported. The plot is not only dominated by the *M2L*-step, but also the *L2L*- and *M2M*-step, which especially increase in dominance through a higher order. The *P2P*-step is almost irrelevant in this configuration as only very small and few blue bars can be seen.

6.4.6. Architectures

Figure 6.8 shows the same parameter combinations as Figure 6.7 but tested on the *Ivy Bridge* platform, described in Subsection 3.3.2, with the yellow line being a test on the *Xeon Phi* platform for reference. Since one *Ivy Bridge* node consists of two processors with eight cores each, only a scaling up to 32 threads, using Hyper-threading, is reasonable.

Comparing the parameter configurations, the same observations can be made as for the *Xeon Phi* analysis. However, when looking at the exact task timings seen in Figure 6.23, a strong shift in computation time from *M2L*- to *P2P*-tasks can be observed. The major difference between these tasks is that the *P2P*-operation tends to be more compute-bound due to its high arithmetic density since it has to compute $O(N^2)$ operations where N is the number of particles in both involved cells. In [Gal16] it is suggested that the vectorized *M2L*-operation is memory bound. Yet, the memory size and bandwidth of the *Xeon Phi* are both higher, which should result in this platform being better suited for performing the *M2L*-operation. From an implementation point of view, a difference in the platforms is that *ls1-mardyn* was linked against different libraries for the Fast Fourier Transformation. For *Ivy Bridge* FFTW[2] was used, while for the *Xeon Phi* the Intel Math Kernel Library (MKL 2017)[3] was used which implements the Fast Fourier Transform with FFTW compatible bindings for *Knights Landing* processors. Further examination of this discrepancy is necessary to fully explain the data.

Comparing the same parameter configurations on the different platforms (purple vs yellow line in Figure 6.8), for the same thread count the time to solution on *Ivy Bridge* is faster by a factor of two, which is consistent with its clock rate being exactly the double of the *Xeon Phi*'s. Only on 32 threads, this pattern is broken, since here the *Ivy Bridge* processors enter Hyper-threading.

6.4.7. Task Timings

When comparing the task timings in Figure 6.10, Figure 6.11, and Figure 6.12, the initial time step takes an increasing share of the overall computation time. This trend is continued when the subdivision factor is increased as seen in Figure 6.16 and even stronger with increased order as seen in Figure 6.17. An explanation for this can be that in this initial step memory allocations are made, which tend to take a long time. Also, as explained in Subsection 5.2.7, in this initial step the tasks are not yet properly weighted, which leads to

[2] `http://www.fftw.org`
[3] `https://software.intel.com/mkl`

suboptimal scheduling. However, this does not impact the scaling plots in Section 6.1 and Section 6.2 since there the initial step is not included.

Taking a closer look at the timing plots of the single time steps, no significant gaps, which would indicate idle time, can be detected, even at the very end of the time steps. This suggests that for the given tasks the scheduling worked very well sustaining a high workload on every processor in spite of the tasks having strongly varying computational cost.

For a comparison with a scheduling featuring gaps and a more detailed analysis of the end of a time step see Appendix A.

6.4.8. Weak Scaling

The purpose of weak scaling is to see how an application behaves when the problem size increases but the relative load on the processing unit stays the same, adding more processors as the size increases. Such a test can be seen in Figure 6.9. The suboptimal performance of the green line suggests that increasing the subdivision factor relatively increases the scheduling overhead of *QuickSched*, while increasing the order has the converse effect, as described above.

All four parameter combinations significantly drop in parallel efficiency at 64 threads. In principle, *QuickSched* was shown to be capable of linear scaling [GCS16] [Gra17], however, for this scenario and shared memory parallelizations of the Fast Multipole Method of *ls1-mardyn* no optimal performance is known.

Nevertheless, the aforementioned drop might indicate that the scheduling overhead does not scale very well for this task pattern and further research is to investigate optimal layout of such patterns.

7. Comparison

Since it is very difficult to compare different implementations of Fast Multipole Methods, as described in Section 2.3, a short comparison to the MPI[1] parallelization of *ls1-mardyn* is made here.

7.1. MPI Version

The MPI parallelization of *ls1-mardyn* is, in contrast to the here presented *QuickSched* parallelization, a distributed memory parallelization. It is based on the idea of splitting the domain into several smaller ones that are each handled by a local multipole cell tree, the so-called local tree. These local trees are then connected by a so-called global tree that propagates the far-field influence of the rest of the domain on each local tree. Furthermore, an exchange of halo layers between the MPI ranks is necessary to fully compute every *M2L*-stencil. A detailed description of the implementation, optimizations, and its performance is made in [Obe16].

Strong scaling experiments with the same parameters as in Figure 6.7 were conducted for pure MPI runs without the *QuickSched* parallelization. In order to preserve comparability, the tests were executed on only one node of the *Xeon Phi* system. However, due to execution limitations, only a scaling up to 32 ranks was possible.

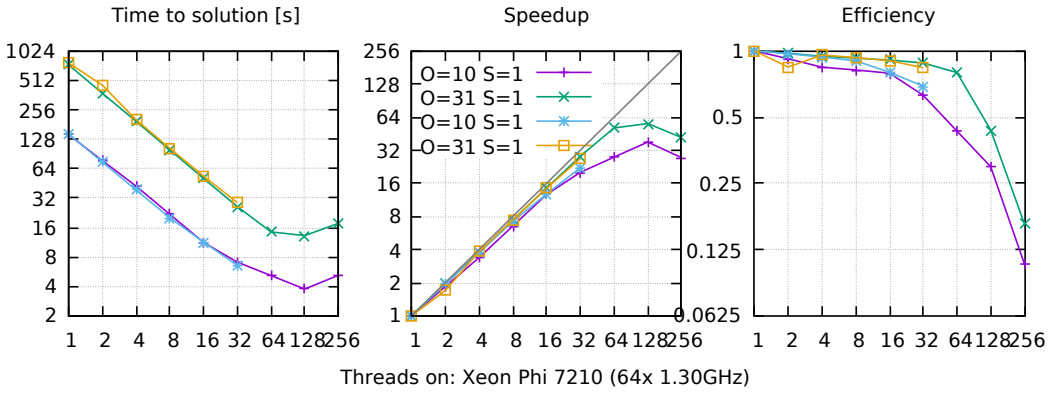


Figure 7.1.: MPI Version: Comparison of different orders at subdivision factor 1, size 80, time steps 30. One rank per thread was used.
Purple and Green are the *QuickSched* version.
Cyan and Yellow are the MPI version.

[1] `http://www.mcs.anl.gov/research/projects/mpi`

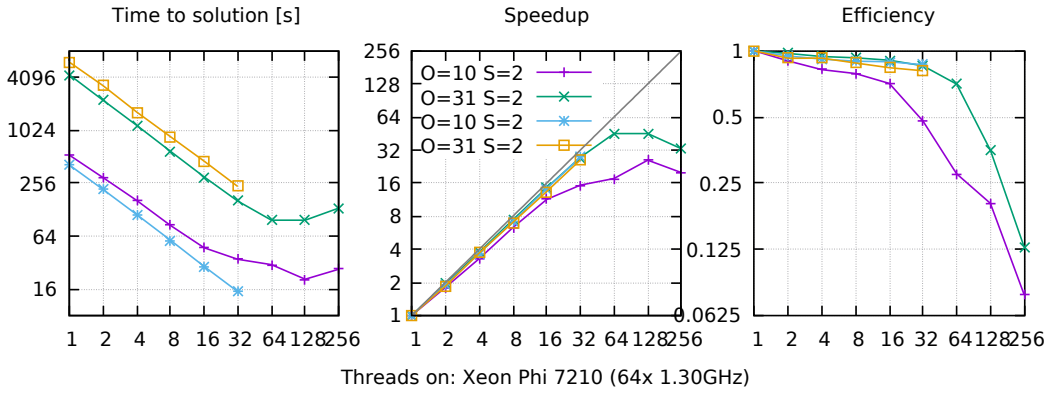


Figure 7.2.: MPI Version: Comparison of different orders at subdivision factor 2, size 80, time steps 30. One rank per thread was used.
Purple and Green are the *QuickSched* version.
Cyan and Yellow are the MPI version.

The strong scaling in Figure 7.1 and Figure 7.2 show good, almost linear speedup throughout the 32 ranks. Also, the parallel efficiency stays above 80% except for the cyan line in the first Figure. This indicates that this parallelization does not suffer the same problems from an increased subdivision factor as the *QuickSched* version.

However, for high orders, the *QuickSched* version outperforms the MPI version. For a subdivision factor of one, the difference is marginally but it increases with subdivision factor of two.

In [Obe16] it is noted that a good load balancing is only given for uniform particle distributions. Since in this scenario the sphere of denser particles was placed exactly in the middle of the domain, the imbalance induced by it was most probably well distributed over several ranks. Moving this sphere and increasing its density or inducing more inhomogeneity to the scenario potentially has a more severe impact on the MPI than on the *QuickSched* version since it has no load balancing mechanism at the moment.

Part IV.

Conclusion

7.2. Summary

A shared memory parallelization of the Fast Multipole Method in *ls1-mardyn* using *QuickSched* was presented. Its scalability potentials were tested and potential problems were explained. It can be concluded that the load balancing of the tasks by *QuickSched* worked well, however, more research is needed to see if the task pattern can be optimized to provide a more linear scaling. At the moment, especially an increase in the subdivision factor, which governs how small all particle cells become, has a grave impact on performance since the computational expense of every single task becomes smaller and their number far greater. The result is a shift of overall computational effort from calculating interactions to scheduling tasks, which harms scalability.

This can be countered by a higher expansion order of the multipole and local expansions, which increases the computational expense per task for all far-field interactions. Thereby, on average, computational effort is shifted back to evaluating interactions instead of scheduling. It was demonstrated that this effect can outweigh the negative effect of the subdivision factor. However, this is not due to a reduction in the computational effort of the scheduling but by substantially increasing the computational effort of evaluating the interactions, which results in a far greater overall time to solution.

On the other hand, the choice of the subdivision factor and order of expansion are parameters which can be chosen perfectly free by the user. A good choice can depend on the particle density, number of Lennard-Jones sites per molecule, desired accuracy, or time to solution. In [Kab12] it is argued that the optimal choice of subdivision factor and order of expansions is reached when the computational effort for the *M2L*- and *P2P*-step are equivalent. This is especially true for the parallelization strategy in this thesis since these steps can be executed completely in parallel. Looking back at the task timing plots in Section 6.3 this would require subdivision factors smaller than one, which is currently not supported by *ls1-mardyn*.

7.3. Outlook

It was demonstrated that through the use of flexible libraries for task based parallelism like *QuickSched* it possible to implement a completely interwoven parallelization of the Fast Multipole Method. A similar approach using *OpenMP 4.5* would have required several barriers and would degenerate to a fork-join approach. However, it needs to be mentioned that according to the latest technical report number 6 (TR6)[2] for the upcoming *OpenMP 5.0* API specification, new mechanisms for more flexibility of explicit tasks are planned, including "mutually exclusive tasks".

Due to the simplicity of *QuickSched* it is easily possible to alter single task types, introduce new ones, and conduct further research on how to improve the task patterns for better scalability. As explained in Section 7.2, reduced scalability is observed when the computational effort per task becomes very small. A way to counter this is by introducing tasks of dynamic size as demonstrated for the Linked-Cell algorithm in [Gra17]. Similar mechanisms can be thought of for the far-field evaluation.

[2] `http://www.openmp.org/press-release/openmp-tr6/`

An interesting point for further investigation would be the cache efficiency of the here presented task patterns to design a more data-driven and cache friendly approach. The current implementation only models dependencies and leaves the rest to the *QuickSched* scheduler, which takes into account which tasks use what resources when distributing them over its queues. Bundling tasks might lead to a decrease in the total amount of tasks and dependencies and improve cache behavior.

Considering that there is now an MPI based distributed memory- and a *QuickSched* / *OpenMP*, based shared memory parallelization, the logical next step would be to merge those two. Such a hybrid parallelization is already thought of in the outlook of [Obe16] and was predicted to be beneficial since it would reduce the number of MPI ranks and thereby lower the amount of communication. This could require adding further *QuickSched* tasks for the MPI communication. Also, every MPI rank could host its own *QuickSched* scheduler, which brings a very dynamic scheduling over the whole simulation.

Finally, it is concluded that the here shown approach is capable of good scaling given the right choices of simulation parameters. Still, further research in optimizing task patterns seems worthwhile.

Part V.

Appendix

A. Observable Effects in Timing Plots

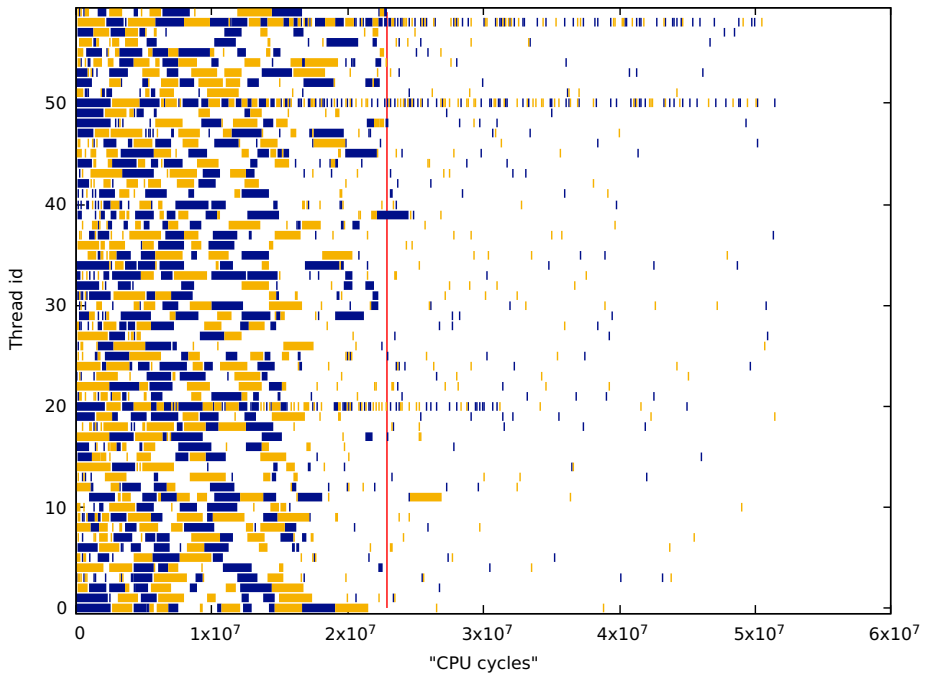


Figure A.1.: Suboptimal task scheduling with large visible gaps left of the red line and scattering of tasks on the right.
Source: [Gra17]

In Figure A.1 two effects can be observed. First, left of the red line, white gaps between the colored tasks can be seen. These stand for time the processor is not executing tasks. They can occur either due to an inefficient scheduling process or because not enough tasks are available at this moment because their dependencies are not yet resolved or they are blocked by other currently executing tasks.

The right-hand side of the red line is mainly white, which means that the processor spends hardly any time on executing tasks. In fact, only the threads with the ids 20, 50, and 58 appear to still execute tasks frequently, which indicates that these are the only threads whose `queues` are still filled. All other threads have empty `queues` and are thus trying to get tasks by work-stealing. However, since the work-stealing implementation works by checking randomly selected `queues` most of the time, empty ones are checked. With a chance of only $\frac{3}{59} \approx 0.05$ to hit a queue that still has tasks, it is reasonable that the majority of threads spend most of their time looking for tasks instead of executing them.

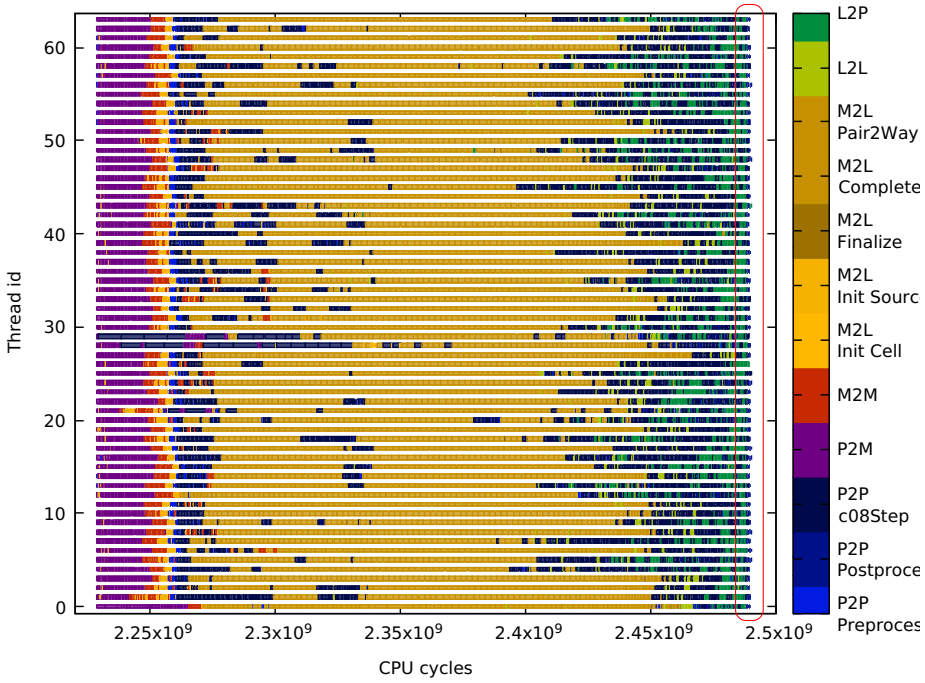


Figure A.2.: A single time step of the simulation explained in Chapter 6 with order 10, subdivision factor 1, and a domain size of 80.

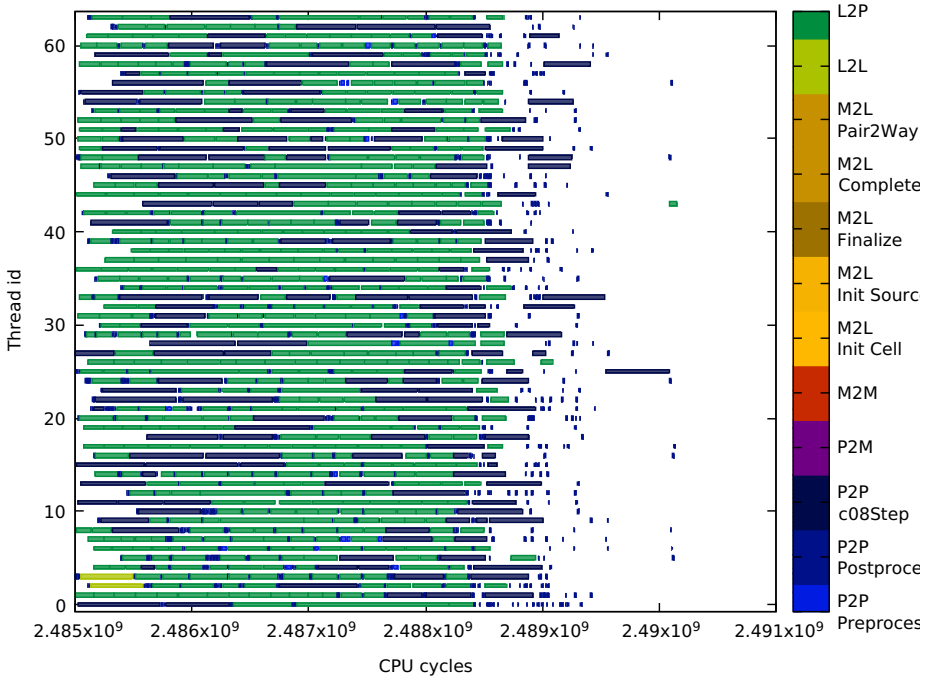


Figure A.3.: Zoom in on the area circled in red in Figure A.2.

In contrast to the situation in Figure A.1, no gaps can be seen in Figure A.2. On the one hand, this is due to a far greater number of tasks, which results in substantially more work available for each thread. On the other hand, Figure A.2 has a length of the $x-$axis of 2.8×10^9 ticks while Figure A.1 only covers 6×10^7 ticks, which makes small gaps in the latter easier visible. However, even when examining Figure A.3, which is a zoom-in on the end of Figure A.2 circled in red, no significant gaps can be found before the very end of the time step. The large white parts on the left side of this Figure are caused by the visualization, which omits tasks that started before the 2.485×10^9-th tick. The whole length the $x-$axis of Figure A.3 are 6×10^6 ticks, which makes it $\frac{1}{10}$-th of the length of Figure A.1. The scattered part starting shortly after 2.488×10^9 has a length of roughly 2×10^6 ticks. Looking at thread 25, 33, and 50, this seems to be approximately the time needed for three consecutive *P2P*-calculation tasks that seem to block each other. Therefore, the size of this scattered end phase appears to be sufficiently small. Also, no heavily imbalanced queues can be observed as in Figure A.1.

List of Figures

List of Tables

Bibliography

[ABC+14] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based fmm for multicore architectures. *SIAM Journal on Scientific Computing*, 36(1):C66–C93, 2014.

[AFH+13] Axel Arnold, Florian Fahrenberger, Christian Holm, Olaf Lenz, Matthias Bolten, Holger Dachsel, Rene Halver, Ivo Kabadshow, Franz Gähler, Frederik Heber, et al. Comparison of scalable fast methods for long-range interactions. *Physical Review E*, 88(6):063308, 2013.

[AMP+13] Abdelhalim Amer, Naoya Maruyama, Miquel Pericàs, Kenjiro Taura, Rio Yokota, and Satoshi Matsuoka. Fork-join and data-driven execution models on multi-core architectures: Case study of the fmm. In *International Supercomputing Conference*, pages 255–266. Springer, 2013.

[BH86] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. *nature*, 324(6096):446–449, 1986.

[Bin95] Kurt Binder. *Monte Carlo and molecular dynamics simulations in polymer science*. Oxford University Press, 1995.

[Cou85] A Coulomb. First memoir on electricity and magnetism. *A Source Book in Physics*, pages 408–413, 1785.

[DES+11] Stephan Deublein, Bernhard Eckl, Jürgen Stoll, Sergey V Lishchuk, Gabriela Guevara-Carrion, Colin W Glass, Thorsten Merker, Martin Bernreuther, Hans Hasse, and Jadran Vrabec. ms2: A molecular simulation tool for thermodynamic properties. *Computer Physics Communications*, 182(11):2350–2367, 2011.

[DYP93] Tom Darden, Darrin York, and Lee Pedersen. Particle mesh ewald: An n log (n) method for ewald sums in large systems. *The Journal of chemical physics*, 98(12):10089–10092, 1993.

[Eck14] Wolfgang Eckhardt. *Efficient HPC Implementations for Large-Scale Molecular Simulation in Process Engineering*. Dissertation, Institut für Informatik, Technische Universität München, München, June 2014. Dissertation erhältlich im Verlag Dr. Hut unter ISBN 978-3-8439-1746-9.

[EHB+13] Wolfgang Eckhardt, Alexander Heinecke, Reinhold Bader, Matthias Brehm, Nicolay Hammer, Herbert Huber, Hans-Georg Kleinhenz, Jadran Vrabec, Hans Hasse, Martin Horsch, et al. 591 tflops multi-trillion particles simulation on supermuc. In *International Supercomputing Conference*, pages 1–12. Springer, 2013.

[EHL80] JW Eastwood, RW Hockney, and DN Lawrence. P3m3dp—the three-dimensional periodic particle-particle/particle-mesh program. *Computer Physics Communications*, 19(2):215–261, 1980.

[Ell95] William Dewey Elliott. Multipole algorithms for molecular dynamics simulation on high performance computers. 1995.

[Ewa21] Paul Peter Ewald. The calculation of optical and electrostatic grid potential. *Ann. Phys*, 64(3):253–287, 1921.

[Fin94] David Fincham. Optimisation of the ewald sum for large systems. *Molecular Simulation*, 13(1):1–9, 1994.

[Gal16] Jean-Matthieu Gallard. Optimization, implementation and evaluation of the fft-accelerated fast multipole method. Master's thesis, April 2016.

[GCS16] Pedro Gonnet, Aidan BG Chalk, and Matthieu Schaller. Quicksched: Task-based parallelism with dependencies and conflicts. *arXiv preprint arXiv:1601.05384*, 2016.

[GGJ11] Christopher G Gray, Keith E Gubbins, and Christopher G Joslin. *Theory of Molecular Fluids: Volume 2: Applications*, volume 10. Oxford University Press, 2011.

[GKZ07] Michael Griebel, Stephan Knapek, and Gerhard Zumbusch. *Numerical simulation in molecular dynamics: numerics, algorithms, parallelization, applications*, volume 5. Springer Science & Business Media, 2007.

[Gon15] Pedro Gonnet. Efficient and scalable algorithms for smoothed particle hydrodynamics on hybrid shared/distributed-memory architectures. *SIAM Journal on Scientific Computing*, 37(1):C95–C121, 2015.

[GR87] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987.

[GR88] Leslie Greengard and Vladimir Rokhlin. The rapid evaluation of potential fields in three dimensions. *Vortex Methods*, 1360:121–141, 1988.

[Gra17] Fabio Gratl. Implementation and evaluation of task-based approaches for molecular dynamics simulations. Studienarbeit/sep/idp, Institut für Informatik, April 2017.

[Kab12] Ivo Kabadshow. *Periodic boundary conditions and the error-controlled fast multipole method*, volume 11. Forschungszentrum Jülich, 2012.

[KDFB04] Douglas B Kitchen, Hélène Decornez, John R Furr, and Jürgen Bajorath. Docking and scoring in virtual screening for drug discovery: methods and applications. *Nature reviews Drug discovery*, 3(11):935–949, 2004.

[KP06] Jakub Kurzak and Bernard M Pettitt. Fast multipole methods for particle dynamics. *Molecular simulation*, 32(10-11):775–790, 2006.

[LJ24] John Edward Lennard-Jones. On the determination of molecular fields. ii. from the equation of state of a gas. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 106, pages 463–477. The Royal Society, 1924.

[LJ31] John Edward Lennard-Jones. Cohesion. *Proceedings of the Physical Society*, 43(5):461, 1931.

[LY14] Hatem Ltaief and Rio Yokota. Data-driven execution of fast multipole methods. *Concurrency and Computation: Practice and Experience*, 26(11):1935–1946, 2014.

[NBB+14] Christoph Niethammer, Stefan Becker, Martin Bernreuther, Martin Buchholz, Wolfgang Eckhardt, Alexander Heinecke, Stephan Werth, Hans-Joachim Bungartz, Colin W Glass, Hans Hasse, et al. ls1 mardyn: The massively parallel molecular dynamics code for large systems. *Journal of chemical theory and computation*, 10(10):4455–4464, 2014.

[NT14] Jun Nakashima and Kenjiro Taura. Massivethreads: A thread library for high productivity languages. In *Concurrent Objects and Beyond*, pages 222–238. Springer, 2014.

[Obe16] Michael Obersteiner. Parallel implementation of the fast multipole method. Master's thesis, October 2016.

[Pet95] Henrik G Petersen. Accuracy and efficiency of the particle mesh ewald method. *The Journal of chemical physics*, 103(9):3668–3679, 1995.

[PG96] EL Pollock and Jim Glosli. Comments on p3m, fmm, and the ewald method for large periodic coulombic systems. *Computer Physics Communications*, 95(2-3):93–110, 1996.

[Pli95] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

[Ran09] David WH Rankin. Crc handbook of chemistry and physics, edited by david r. lide, 2009.

[TNYM12] Kenjiro Taura, Jun Nakashima, Rio Yokota, and Naoya Maruyama. A task parallel implementation of fast multipole methods. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 617–625. IEEE, 2012.

[TWG+15] Nikola Tchipev, Amer Wafai, Colin W Glass, Wolfgang Eckhardt, Alexander Heinecke, Hans-Joachim Bungartz, and Philipp Neumann. Optimized force calculation in molecular dynamics simulations for the intel xeon phi. In *European Conference on Parallel Processing*, pages 774–785. Springer, 2015.

[vGB90] Wilfred F van Gunsteren and Herman JC Berendsen. Computer simulation of molecular dynamics: Methodology, applications, and perspectives in chemistry. *Angewandte Chemie International Edition in English*, 29(9):992–1023, 1990.

[YB12] Rio Yokota and Lorena A Barba. Hierarchical n-body simulations with autotuning for heterogeneous systems. *Computing in Science & Engineering*, 14(3):30–39, 2012.

[Yok13] Rio Yokota. An fmm based on dual tree traversal for many-core architectures. *Journal of Algorithms & Computational Technology*, 7(3):301–324, 2013.