technische universität dortmund

Diplomarbeit

Engineering the Fast-Multipole-Multilevel Method for multicore and SIMD architectures

Martin Gronemann

26. Februar 2009

Betreuer:
Prof. Dr. Petra Mutzel
Dipl.-Inform. Carsten Gutwenger

Fakultät für Informatik
Algorithm Engineering (Ls11)
Technische Universität Dortmund
http://ls11-www.cs.uni-dortmund.de

Acknowledgment

The reader may forgive me for writing this in German:

An dieser Stelle möchte ich mich bei all denen bedanken, die mich während meiner Studienzeit begleitet haben. Als erstes danke ich meinen beiden Betreuern, Prof. Dr. Petra Mutzel und Carsten Gutwenger, nicht nur für die Betreuung dieser Arbeit, sondern auch dafür, dass sie mir überhaupt erst das Gebiet des Graphenzeichnens nähergebracht haben. Weiterhin möchte ich meinen Eltern danken, die mir das Studium ermöglicht haben und mich währenddessen immer unterstützt haben. Ganz besonders möchte ich mich bei Bernd Zey für die unzähligen Diskussionen, seine Geduld und die gemeinsame Studienzeit bedanken.

Short Abstract

In this thesis, we present a new variant of the *Fast-Mulitpole-Multilevel Method*, which is used to draw large graphs. Based on the original approach by Stefan Hachul, a new algorithm is presented, which is optimized primarily for practical speed. In order to achieve this, special processor instructions are used to accelerate computations with complex numbers. In addition, parts of the algorithm are executed in parallel to benefit from the widely spread multicore architectures. Besides these two rather technical improvements, we describe a new construction method for a spatial space decomposition data structure, called the *quadtree*. The algorithm exploits the binary representation of the coordinates and shifts most of the work to the sorting of the input. Furthermore, we introduce another problem from computational geometry, the *well-separated pair decomposition*, and successfully apply it in order to simplify parts of the algorithm. The resulting algorithm is able to compete in speed and layout quality even with a recently published graphics processor accelerated implementation.

Zusammenfassung

Diese Diplomarbeit stellt einen neuen Entwurf der *Fast-Mulitpole-Multilevel Methode* vor, die zum Zeichnen von großen Graphen verwendet wird. Basierend auf dem ursprünglichen Algorithmus von Stefan Hachul wird eine neue Variante vorgestellt, die besonderes im Hinblick auf die praktische Rechenzeit optimiert ist. Dabei wird zum einen von speziellen Intruktionen Gebrauch gemacht, um die Berechnungen mit komplexen Zahlen zu beschleunigen. Zum anderen werden Teile des Algorithmus parallel ausgeführt, um die mittlerweile weit verbreiteten Multicore Prozessoren effizienter nutzen zu können. Neben diesen beiden relativ technischen Verbesserungen wird ein neuer Ansatz zur Berechnung einer Raumunterteilungshierachie, den sogenannten *Quadtrees*, vorgestellt. Dieser macht sich diverse Eigenschaften der Zahlendarstellung im Binärformat zu Nutze und verlagert einen Großteil der Problematik auf das Sortieren der Eingabe. Zusätzlich wird ein weiteres Problem aus der algorithmischen Geometrie, die *Well-Separated Pair Decomposition*, verwendet um diverse Teile des Algorithmus zu vereinfachen. Der resultierende Algorithmus kann im Bezug auf Geschwindigkeit und Qualität mit einer krlich veröffentlichten, auf Grafikkarten basierenden Implementierung mithalten.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 26. Februar 2009

Martin Gronemann

Contents

Chapter 1

Introduction

Graphs are an abstract way to represent all kinds of relations between entities. Entities are encoded as a set of *nodes*, and relations between two entities are expressed by *edges*. Therefore, in practice graphs are often used as a tool for analyzing and visualizing such relationships.

For example a road map can be displayed as a graph by assigning each junction a node and road sections are represented by edges connecting the corresponding nodes. If one is interested in the shortest path between two places a generic graph algorithm can be applied. However, when such relations are analyzed by a human instead of an algorithm, the problem arises to draw the graph. The road map is a drawing itself with predefined positions of roads and cities, which lies in the nature of things. But when considering a computer network, the reader might be interested in the structure of the network, rather than the physical positions of the computers in some server room or office.

This leads to the problem of calculating a drawing (or layout), based on the structural information provided by the graph itself. Since the concept of a graph can be applied in many ways, the drawing of graphs is used in various areas within computer science, as well as in other fields. Common application areas are:

- *Software Engineering:* Unified Modeling Language Diagrams (UML)
- *Database Development:* Structured-Entity-Relationship-Model (SERM)
- *Automation Engineering:* Finite State Machines (FSM)
- *Biology:* Protein Interaction Networks

The drawing is usually done in the plane, although there exists layouts for three dimensions. However, in the following we focus only on drawings in two dimensions.

Regardless of the application area, the drawing of a graph should be nice to read and reflect the structure of the graph. Therefore, it is necessary to formulate *aesthetic* *criteria*, which define a good drawing. The following common properties are proposed in [Pur97, PAC01] and investigated there during empirical studies:

- *Minimize Bends:* The number of bends should be minimized.
- *Orthogonality:* Fix nodes and edges to an orthogonal grid.
- *Number of Edge Crossings:* Usually less crossings are considered to be better because it is easier to follow an edge from one node to another.
- *Size of the Drawing:* The area covered by the layout should be as small as possible.
- *Edge Length:* The lines used to represent the edges should be as short as possible but still long enough to allow the nodes to be clearly separated.
- *Symmetry:* If the graph is symmetric the drawing should also reflect these symmetries.

Some of these rules conflict with each other like, e. g., the minimization of edge crossings and bends in an orthogonal layout. In order to reduce the crossings, sometimes it is necessary to insert more bends. Furthermore, the problem of crossing minimization is an NP-hard problem.

Dependent on the application area, some criteria are more important than others. As a result, different kinds of layouts with different properties have been developed. An overview about the various layouts can be found in [JM04].

Apart from the different layouts and the induced problems, the applicability of a layout algorithm depends on two other aspects:

Runtime: In some application areas like, e. g., in bioinformatics, very large datasets have to be visualized. Therefore, a fast algorithm is preferred to an algorithm, which may produce a better layout.

Structural Requirements: Some algorithms are only applicable if the graph meets some condition, like, e. g., planarity (the graph can be drawn without crossings). If the input does not satisfy these requirements, it has to be modified (in case of planarity, the graph is planarized).

After this short introduction to graph drawing, we give a brief introduction to *force-directed* graph drawing and some previous work. The following chapters require only basic knowledge in graph theory, thus no introduction is given. Other preliminaries needed are introduced in the corresponding chapters for reasons of clarity. Since our algorithm makes use of parallel computation, we introduce the concept of parallel machines in general, and at the end of this chapter, the objective and basic structure of the thesis is outlined.

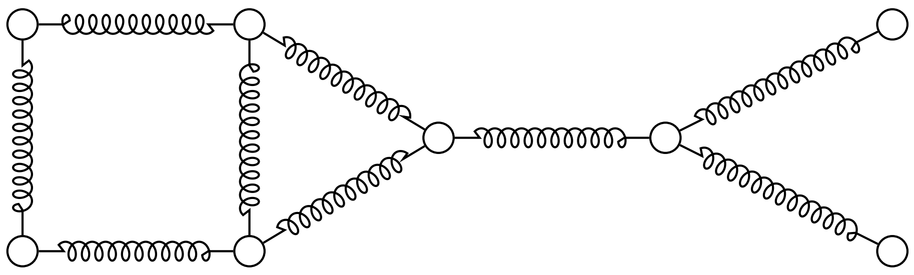


Figure 1.1: Principle of a force-directed layout algorithm.

1.1 Force-Directed Graph Drawing

In this section, the basic principles of the force-directed layout method are presented. Tutte [Tut63] marks the starting point of both graph drawing in general and the force-directed layout.

The basic idea of the force-directed layout is to run a physical simulation on a given drawing of a graph, where each pair of nodes repel each other and edges act like springs (Figure 1.1). The springs are used to keep adjacent nodes close to each other, whereas the repulsive forces will unfold the drawing.

The simulation is done by repetitively calculating the forces acting on the nodes and moving them accordingly. However, the method uses a simplified physical model, because values like mass, friction, velocity, and acceleration are not taken into account. Instead, the forces are used directly as a displacement vector at the end of each iteration. Before going into detail, we make an agreement about the usage of variables, representing a scalar or a vector, when it comes to forces: F describes an amount of force whereas $\vec{F}$ describes a force vector.

In each iteration of the simulation, the total force $\vec{F}_{\text{res}}(v)$ for each node $v \in V$ has to be calculated, consisting of repulsive and attractive forces acting on v:

$$\vec{F}_{\text{res}}(v) = \vec{F}_{\text{rep}}(v) + \vec{F}_{\text{attr}}(v)$$

Since both the repulsive and the attractive force function only depend on the distance and not on the direction of the other node, they can be defined as a scalar function, mapping the distance to a scalar force value. Regardless of the exact definition of the repulsive force function F_{rep}, the repulsive force vector for a node v can be obtained by

$$\vec{F}_{\text{rep}}(v) = \sum_{w \in V \setminus \{v\}} \frac{\vec{p}_v - \vec{p}_w}{|\vec{p}_v - \vec{p}_w|} \cdot F_{\text{rep}}(|\vec{p}_v - \vec{p}_w|)$$

and the attractive forces due to all incident edges of v by

$$\vec{F}_{\text{attr}}(v) = \sum_{e=(v,w) \in E} \frac{\vec{p}_v - \vec{p}_w}{|\vec{p}_v - \vec{p}_w|} \cdot F_{\text{attr}}(|\vec{p}_v - \vec{p}_w|)$$

Algorithm 1.1 Force-Directed Layout Algorithm

```
1: function SpringEmbedder($G, p, \lambda, i_{\max}, \Delta_{\text{threshold}}$)
2:     $i \leftarrow 0$
3:     repeat
4:         $\Delta_{\max} \leftarrow 0$
5:         for $v \in V$ do
6:             $\vec{F}_{\text{res}}(v) \leftarrow \vec{F}_{\text{rep}}(v) + \vec{F}_{\text{attr}}(v)$
7:         end for
8:         for $v \in V$ do
9:             $\vec{d_v} \leftarrow \lambda \cdot \vec{F}_{\text{res}}(v)$
10:            $\Delta_{\max} \leftarrow \max(\Delta_{\max}, |\vec{d_v}|)$
11:            $\vec{p_v} \leftarrow \vec{p_v} + \vec{d_v}$
12:        end for
13:        $i \leftarrow i + 1$
14:    until $\Delta_{\max} < \Delta_{\text{threshold}}$ or $i \geq i_{\max}$
15: end function
```

where F_{attr} is the edge force function. When the force vectors for all nodes are calculated, the nodes can be moved. Therefore, the force vector is multiplied by a constant $\lambda > 0$ and then added to the current coordinates of the node. The factor λ is called the *time step*.

$$\vec{p_v} := \vec{p_v} + \lambda \cdot \vec{F}_{\text{res}}(v)$$

This procedure is repeated until a maximum number of iterations $i_{\max}$ is done or the maximum displacement length falls under a given threshold $\Delta_{\text{threshold}}$. The objective is to find a placement at which the nodes are in a *state of equilibrium*. This state is reached, when the total force acting on all nodes is zero.

Definition 1.1.1. *Given a graph $G = (V, E)$ and a node placement $P = \{p_v \in \mathbb{R}^2 \mid v \in V\}$, the nodes are in a state of equilibrium iff*

$$\forall v \in V : |\vec{F}_{\text{res}}(v)| = 0$$

Algorithm 1.1 describes the framework in detail. The running time of Algorithm 1.1 is dominated by the force calculation in line 6, since the rest can obviously be done in $O(|V|)$. First, the attractive forces induced by the edges can be evaluated in time $O(|E|)$. It remains the repulsive force calculation. The exact approach would have to consider every ordered pair of nodes, which requires $O(|V|^2)$ time. Thus, the time needed for one iteration is $O(|V|^2 + |E|)$ and the total running time is $O(i_{\max} \cdot (|V|^2 + |E|))$. In order to use the algorithm for calculating layouts for large graphs, the running time has to be improved in two points:

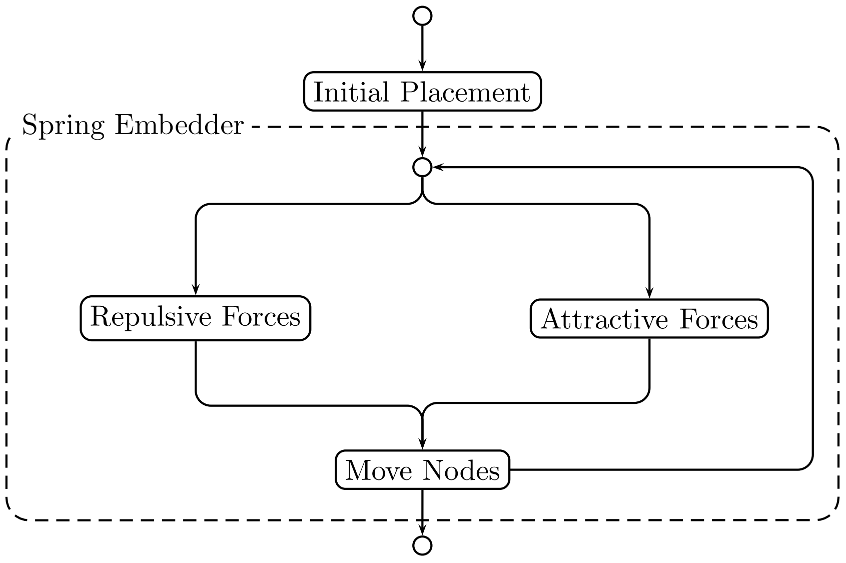


Figure 1.2: Basic excerpt of a force-directed layout algorithm

1. The time needed for the repulsive force calculation has to become sub-quadratic.
2. The number of iterations needed should be as small as possible.

However, the basic principle all force-directed algorithms share is displayed in Figure 1.2. In the next chapters we will extend this diagram by adding more steps to it.

Next, the choice of the force function is discussed and some previous work is presented. First, we introduce the force model of Eades [Ead84]. Let $d = |\vec{p}_u - \vec{p}_v|$ denote the distance between the nodes u and v. The repulsive and attractive force functions are defined there as

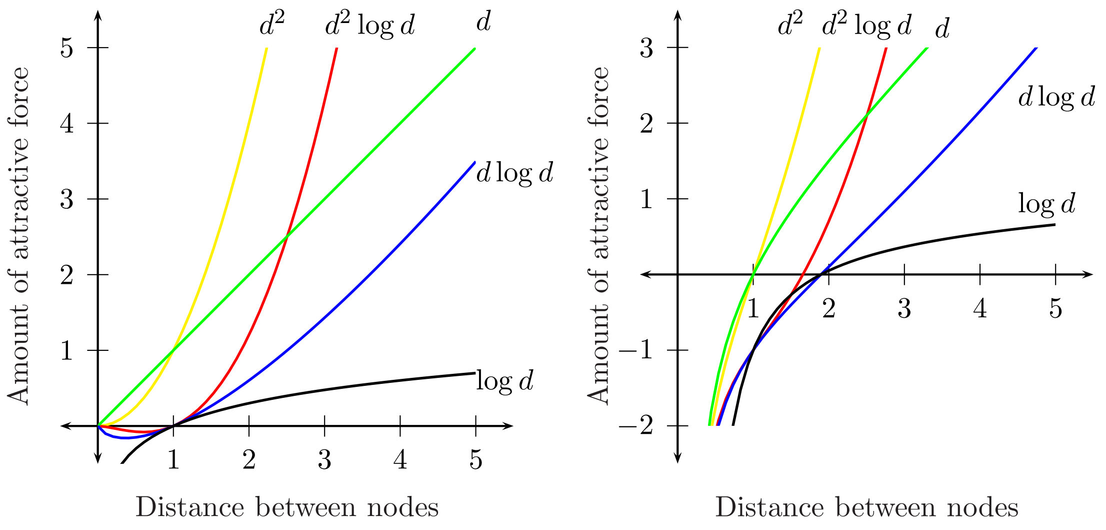
$$F_{\text{rep}}^{\text{Eades}}(d) = \frac{c_r}{d^2} \quad \text{and} \quad F_{\text{attr}}^{\text{Eades}}(d) = -c_a \log d,$$

where c_r and c_a are two constants. Fruchterman and Reingold [FR91] introduced a set of force models and a grid based approach to reduce the complexity of the repulsive force calculation step. The so called *Grid-Variant Algorithm* uses a grid of squares and assigns each node a square. Then the repulsive forces for a node are only computed for nearby squares. The basic idea is, since most of the repulsive force functions are rapidly decaying with increasing distance, that the error is small when skipping these. The time needed for this approximation is $O(|V|)$, assuming the nodes are uniformly distributed over the grid squares. Besides the force model of Eades, they experimented with two other functions, starting with

$$F_{\text{rep}}^{\text{FR1}}(d) = \frac{c^2}{d} \quad \text{and} \quad F_{\text{attr}}^{\text{FR1}}(d) = -\frac{d^2}{c}.$$

Furthermore, they tried

$$F_{\text{rep}}^{\text{FR2}}(d) = \frac{c}{d} \quad \text{and} \quad F_{\text{attr}}^{\text{FR2}}(d) = -\frac{d}{c}.$$



(a) Attractive force due to the spring length. (b) Resulting attractive force due to both spring and repulsive force.

Figure 1.3: Comparision of the various force functions. In (a) the amount of attractive force by edge length is shown, whereas in (b) the repulsive force ($1/d$ except Eades with $1/d^2$) is subtracted.

Hachul's *Fast Mulitpole Mulltilevel Method*(FM^3) [Hac05] uses basically the same repulsive force function, whereas the attractive force function can be parametrized with the desired length d_e of the edge:

$$F_{\text{rep}}^{\text{Hachul}}(d) = \frac{c_r}{d} \quad \text{and} \quad F_{\text{attr}}^{\text{Hachul}}(d) = -c_a \cdot \log \frac{d}{d_e} \cdot d^2$$

For the repulsive force calculation, an approximation is used called the *Fast Multipole Method*. The basic idea is to approximate the forces due to nodes located far away by grouping them together and using a power series describing the forces. Since this work is heavily based on Hachul's work, especially on his multipole framework, we choose the same repulsive function. The multipole framework will be dealt with in detail in a later chapter. The attractive force function is modified, to avoid the need of stabilizing measures:

$$F_{\text{rep}}(d) = \frac{c_r}{d} \quad \text{and} \quad F_{\text{attr}}(d) = -c_a \cdot \log \frac{d}{d_e} \cdot \frac{d}{\deg(v)}$$

As an example how these functions behave, assume two adjacent nodes are given. Dependending on the edge length alone, the amount of attractive force is shown in Figure 1.3(a), whereas (b) displays the resulting attractive force when, additionally, the repulsive force of the other node is taken into account. The desired edge length of the two last mentioned functions has been set to 1.0. Clearly visible in (a) is the sign change due to the use of the logarithm at a distance equal to the desired edge length. When taking the repulsive forces into account, the problem arises that the amount of attractive forces is zero when the edge is nearly two times (1.5 times respectivly) longer then the desired edge length. However, the scenario of two nodes is a simple one. Consider an edge connecting two large sets of nodes alone, then the edge has to counter the repulsive forces between the two sets, and not only between the two incident nodes. When a state of equilibrium is reached, there is no guarantee for complying with the desired edge length, at best the value acts as guidance. However, the final edge length dependents on the graph structure and how much repulsive force the edge has to counter.

In order to reduce the number of steps needed during the simulation, two things can be done. First, the time step is increased as much as possible without affecting the stability. Increasing the time step results roughly in fewer iterations. Since we are not interested in what is happening during the simulation, rather than in the result, the simulations should run as fast as possible, regardless of the errors due to a bigger time step.

The second measure to reduce the number of iterations is a good initial placement for the nodes. Obviously, a random placement needs more steps than a placement which is already close to the desired result. This leads to the *multilevel* method, which generates multiple graphs $G_1, \ldots, G_m$ with different resolutions from the original graph $G = G_0$. Then the layout for the different levels is calculated, starting with the coarsest one G_m. In each step, the initial placement for G_t is calculated by the positioning of the nodes in the next coarser level G_{t+1}.

1.2 Parallel Machines

In this section, an introduction to the concepts of parallel machines is given with special regards to the ones used in this thesis. First, *shared memory machines* are introduced, since they are the underlying concept of modern multicore architectures. Then, the *parallel data* concept is presented, which is used in form of the *Single-Instruction-Multiple-Data (SIMD)* extensions contained in today's processors.

Starting with the mainframes in the early 1960's, the shared memory multiprocessor system is one of the most important classes of parallel machines. The key property of these systems is the ability to do implicit communications through shared memory space with simple read and write operations. The main memory can be used like a pin board, where one job leaves a note for another one. If the costs for accessing a memory location is the same for all processors, then the system is called a *symmetric multiprocessor (SMP)* system. Usually in an SMP all processors are connected by a bus to the main memory or other components like IO controllers (see Figure 1.4).

The programming model of this system does not rely on the number of processors. Instead, a processor is able to handle multiple jobs through time sharing, where each job gets a specific amount of time assigned to do its work. In general, these jobs are called *threads*, where each thread belongs to the owning process. Formally, a *process* consists of a shared address space and at least one thread of control [CwAG99]. The first thread

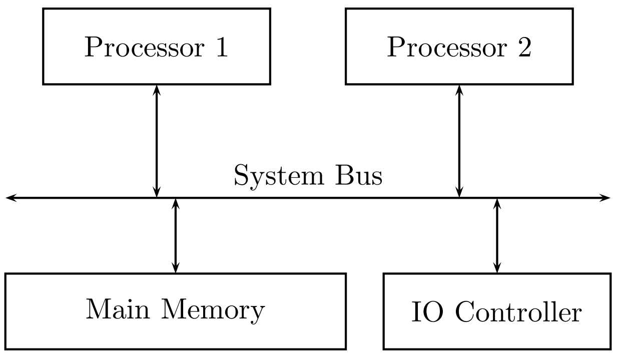


Figure 1.4: Two symmetric multiprocessors and the main memory

is the *main (control) thread.* If the process has at any time only this one thread it is called *single-threaded.* Otherwise, the process is called *multi-threaded* and the threads share the address space of the process. Note that threads belonging to different processes do not share the same memory address space and therefore communication is restricted to methods referred to as "Inter-Process communication". Threads can be created at any point and even one thread can create a new one. If the machine contains multiple physical processors, then a multi-threaded process (or application) can benefit from it by creating threads, so the operating system can assign each processor a thread in order to execute code in parallel, whereas in single-threaded applications, only one thread is available and therefore only one processor can be used. In order to do work in parallel using threads some aspects have to be considered.

- Platform (architecture, operating system and thread programming interface)
- Basic primitives for synchronization (and their overhead)
- Parallelization of the algorithm

In this thesis we target the IA-32 and Intel 64 architecture with multiple processors/cores running Microsoft Windows and/or Unix based operating systems (like Linux, BSD, Mac OS X). Therefore, we first discuss the available options of thread API's. Thread programming interfaces are usually provided by the operating system itself or an external library. For example, Microsoft Windows has its own thread API, Linux provides kernel functions but most Unix-like systems have the posix thread library installed, which provides a unified interface for thread programming for Unix based systems. Other external libraries are Intel Threading Building Blocks, OpenMP, Boost (includes platform independent thread support), and NVidia's CUDA for GPUs. Since the algorithm is part of the *Open Graph Drawing Framework (OGDF)* [OGD], it is required to run on all of

OGDF's target platforms. In our case we will use a thread class already implemented in OGDF, which uses pthreads and the Windows API and continue using these two APIs when additional functionality is required.

When it comes to execute code in parallel, it is necessary to synchronize it at some point. The classic example for this is a variable used by two threads at the same time. In order to change a piece of memory, the system has to load it from the address into a register, do a computation, and store it back to memory. Consider two threads, incrementing an integer with a value of zero. Obviously, the wanted result is two. When the second thread loads the integer before the first thread has stored it, they both will increment it in their register, and store it back to the main memory. As a result, the memory location contains the value of the last storing thread, thus, the resulting value is one.

In order to avoid accessing a shared resource at the same time, a synchronization primitive called *critical section* (sometimes referred to as *mutex* or *lock*) can be used. A critical section marks a piece of code that can only be executed by one thread at the same time. In the previous example, the problem could be solved by letting a thread enter the critical section before using the integer, and leaving it when finished, thereby guaranteeing exclusive access to the integer when incrementing it.

Another useful primitive is the *condition variable*. Basically, a thread can either wait for the condition or trigger it, in order to signal other threads to continue. In this thesis the two primitives are not used directly; instead they are an essential part of a mechanism to synchronize threads called a *barrier*. A barrier marks a kind of checkpoint all threads have to reach before they are allowed to continue. The only rule to obey, when using barriers, is that each thread must be able to reach this checkpoint. For example, if a barrier is used inside an if-then-else statement, all threads have to either use the barrier or not. Otherwise some threads are waiting for an event that will never occur. Apart from this rule, barriers represent an easy way to synchronize threads, especially when they execute the same code just with different parameters.

Furthermore, we introduce another concept, the *data parallel processing*. Assume an array of data elements is given and the same instruction has to be executed for each element. Usually the elements are processed sequentially. In older computer systems, fetching an instruction was very expensive and therefore the concept of *single instruction multiple data (SIMD)* arised. An SIMD capable processor can perform one instruction on a fixed number of elements in parallel and therefore reducing the amount of instructions. This is still a common concept in modern CPUs and GPUs (graphics processing units) and very useful when it comes to vector calculations.

In order to benefit from an SIMD capable processor, Intel's SSE (Streaming SIMD Extensions) are used via intrinsics. Intrinsics are a lot more comfortable than using SSE in assembler, since an instruction can be used like a C-function and does not require any inline assembler blocks. Furthermore, the user is relieved from managing the physical

```
// two 16 byte aligned pointer
double* ptr1, ptr2;
// two 128 bit register
__m128d reg1, reg2;
// load two double precision values
reg1 = _mm_load_pd(ptr1);
// reg1 contains ptr1[0] and ptr1[1]
reg2 = _mm_load_pd(ptr2);
// reg2 contains ptr2[0] and ptr2[1]
reg1 = _mm_mul_pd(reg1, reg2);
// reg1 contains ptr1[0]*ptr2[0] and ptr1[1]*ptr2[1]
_mm_store_pd(ptr1, reg1);
// ptr[0] := ptr1[0]*ptr2[0]
// ptr[1] := ptr1[1]*ptr2[1]
```

Listing 1.1: Example for using Intel's SSE to multiply double precision values

available registers, because a 128-bit register can be introduced like a standard variable and the compiler will deal with the register assignment. Listing 1.1 shows a small example how to load, multiply, and store two double precision floating point numbers in parallel. The `_mm_load_pd` statement loads two double values into a register; the address must be aligned on a 16-byte boundary. After loading a second pair into the other register the pairs are multiplied by `_mm_mul_pd` and the result is stored in the first register. Finally `_mm_store_pd` writes the result pair back to main memory. For a complete reference of all intrinsics available see for example [Int08a, Int08b].

1.3 Objectives and Outline

The objective of this thesis is to engineer the fast multipole multilevel method for multicore and SIMD architectures, based on the work of Hachul [Hac05]. Instead of optimizing the original implementation, a new algorithm has been implemented which follows the basic ideas of the original one.

First, we introduce in Chapter 2 the *Fast Multipole Method*, originally invented by Greengard [GR87] and improved by Hachul [Hac05, HJ04]. Furthermore, we describe a simple way to accelerate all calculations involving complex numbers by using Intel's Streaming SIMD Extensions.

In Chapter 3, a new algorithm for constructing a quadtree is developed, which is more optimized for speed rather than having a provable good theoretic running time for all possible distribution of points. Instead of using a traditional top-down approach, a bottom-up construction is presented which is quite technical, but results in a fast implementation. Furthermore, we describe an efficient memory layout for the data structure.

The presented quadtree construction lacks an ability, which is required for applying the Fast Multipole Method. Thus, another problem of computational geometry, the *Well-Separated Pair Decomposition (WSPD)* is introduced in Chapter 4 and successfully applied.

All previously described parts are then used in Chapter 5, where a force-directed layout algorithm named the *Fast Multipole Embedder (FME)* is developed. In order to benefit from multicore architectures, a parallel version of the algorithm is described as well.

The concept of multilevel layout algorithms is used in Chapter 6 together with the FME to obtain the *Fast Multipole Multilevel Embedder (FMME)*. Figure 1.5 shows how the aforementioned chapters are used to describe the different parts inside the *Fast Multipole Multilevel Embedder*.

The algorithm has been implemented in C++ as part of the Open Graph Drawing Framework [OGD]. In Chapter 7, we present the practical runtime for various benchmark graphs on a Dual Socket Intel Xeon E5430 system with eight cores and compare it with the original implementation and a graphics card based implementation by Godiyal et al. [GHGH09]. Furthermore, the resulting layouts are given at the end of the chapter.

The last chapter, summarizes the results of this thesis and we give some ideas what can be done in the future.

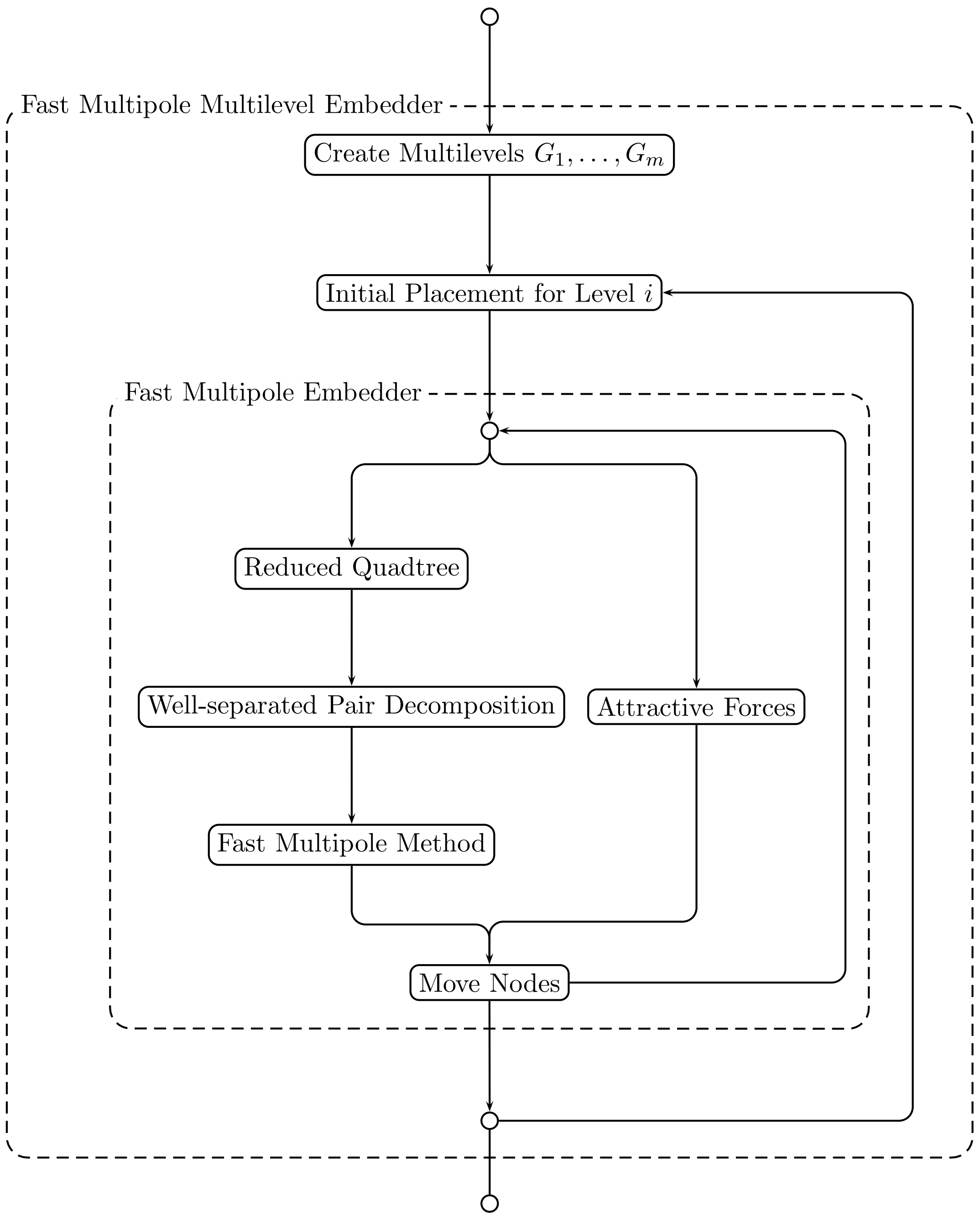

Fast Multipole Multilevel Embedder
Create Multilevels $G_1, \ldots, G_m$
Initial Placement for Level i
Fast Multipole Embedder
Reduced Quadtree
Well-separated Pair Decomposition
Attractive Forces
Fast Multipole Method
Move Nodes

Figure 1.5: Basic excerpt of our Fast Multipole Multilevel Method

Chapter 2

The Fast Multipole Method

In this chapter, we describe the Fast Multipole Method, its applications, and finally some implementation details. The method was invented by Greengard and Rokhlin [GR87] and is used to approximate the forces acting in an N-Body system with gravitational or coulombic forces.

In an N-body system, a set of n bodies repel or attract each other by a given force function. The task is to calculate the acting forces on each of the objects or at some other specified positions. Then, e. g., the forces can be applied in order to find the trajectory for each body in a given time interval. These simulations have a wide range of applications and can be used in many different ways. In our case, the bodies are the graph nodes and we want to evaluate the forces between them during each iteration of the force-directed layout algorithm.

The arising problem is the time needed to evaluate the functions. Assume n particles are given and we want to calculate the forces acting on each particle due to the other $n-1$ particles. Then the naive approach has a running time of $O(n^2)$, because each ordered pair of particles would be considered. The running time is obviously not practical for large n.

In the following, we will first give two examples of other application areas, starting with gravitational systems, the base for the simulation of celestial bodies. For the physical background, we refer the reader to [HRW05]. In a gravitational system, two bodies with mass m_i and m_j attract each other by an amount of force

$$F_{ij}^G = \gamma \frac{m_i m_j}{|p_i - p_j|^2},$$

where γ is the gravitational constant. For example, the *Millennium Simulation Project* [mil] at the Max-Planck-Institut für Astrophysik in Garchingen, Germany, used more than ten billion particles in their simulation, which kept the supercomputers busy for one month. The objective was to simulate the forming of galaxies to gain more knowledge how these formations were created from the early beginnings of the universe.

Another application area is the simulation of electrostatic fields created by charged particles. Given two particles with charge q_i and q_j, the amount of electrical force acting between them due to their charge is defined by the law of Coulomb as

$$F_{ij}^C = \frac{1}{4\pi\epsilon_0}\frac{q_i q_j}{|p_i - p_j|^2},$$

where q_i is the charge of particle i (respectively j) and ε_0 is the permittivity.

Both force functions are of the form $1/d^2$ and therefore rapidly decaying when the distance d between the objects increases. However, even at large distances this force cannot be ignored, because, e. g., a large cluster of particles can create a significant force even over a long distance. The difference between the two functions is, of course, the direction of the force vector. In a gravitational system two bodies attract each other, whereas in a coulombic system this is only the case when the two particles have a positive and a negative charge.

2.1 Force, Potential Fields and Energy

In the following, we use terms derived from electrostatics. However in this thesis the Fast Multipole Method is used to serve our purposes in graph drawing and therefore we will deviate from the principles of physics. Since we are interested in repelling forces to simulate the repulsion between nodes, we will use a force function similar to the coulombic version and assume the charges are both positive. Like in [Hac05] we choose F_{ij} for two particles at distance d as

$$F_{ij}(d) = \frac{q_i q_j}{d}$$

and the force vector respectively as

$$\vec{F}_{ij} = F_{ij}(|p_i - p_j|) \cdot \frac{p_i - p_j}{|p_i - p_j|}.$$

Given one particle i, the electric field induced by this particle is

$$\vec{E}_i(x) = \frac{q_i \cdot (x - p_i)}{|x - p_i|^2}$$

Note that $\vec{E}_i$ is a vector field and it depends only on the creating particle. To obtain the forces acting on a particle due to the electric field of another particle, we use the relation between the electric field and the force:

$$\vec{F}_{ij} = q_i \cdot \vec{E}_j(p_i).$$

Given n particles, since we sum up the forces, we can sum up the electric fields to obtain one vector field for all particles.

$$\vec{E}(x) = \sum_{i=1}^{n} \vec{E}_i(x)$$

The idea of the Fast Multipole Method and other approximation methods is to divide the sum into two parts:

$$\vec{E} = \vec{E}_{\text{far}} + \vec{E}_{\text{near}}$$

The first part called $\vec{E}_{\text{far}}$ is the force field due to particles located far enough away from the point of analyticity. The Fast Multipole Method is used to approximate that function. On the other hand $\vec{E}_{\text{near}}$ describes the forces induced by particles close to the point of analyticity, which are evaluated directly. Therefore, in our application we have to determine for each graph node a set of near and far nodes. However, we first present the Fast Multipole Method which is used to approximate $\vec{E}_{\text{far}}$.

In the following, a small introduction to potential energy and fields is given. Everything is based on our force model and not on the law of Coulomb. See [HRW05] for a more detailed introduction to this matter.

Assume a fixed particle i with charge q_i is given and we can move a test particle j with charge q_j to different positions relative to the first particle. At first the test charge is located at distance d_0 from i. Now we move the test charge j towards the other particle to a distance of d. Then the potential energy of the test charge increases by the amount of work needed to move it against the repelling force.

$$V_{ij} = -\int_{d}^{d_0} F_{ij}(s)ds = q_i \cdot q_j \cdot (\log(d) - \log(d_0))$$

Since the test particle has a charge itself, the potential energy between two particles depends on both particles. However, like the electrical field, the potential field of a particle depends only on the particle itself. Therefore we define the potential field of a particle like in [Hac05] as

$$\Phi_i(x) = -q_i \cdot \log(|x - p_i|).$$

This leads to the simple rule: When moving against the electrical field, the potential energy increases, when moving with the electrical field the potential energy decreases. Figure 2.1 shows an example of two particles with the same charge, their electric force field, and the resulting potential field.

To prepare for the multipole expansion theorem in the next section, we refer from now on to a point $(x, y) \in \mathbb{R}^2$ as the complex number $z = (x + iy) \in \mathbb{C}$ and introduce the complex function $\mathcal{E}$, like in [Hac05], as the potential energy function due to a charged particle i located at p_i with charge q_i:

$$\mathcal{E}(z) = q_i \cdot \log(z - p_i)$$

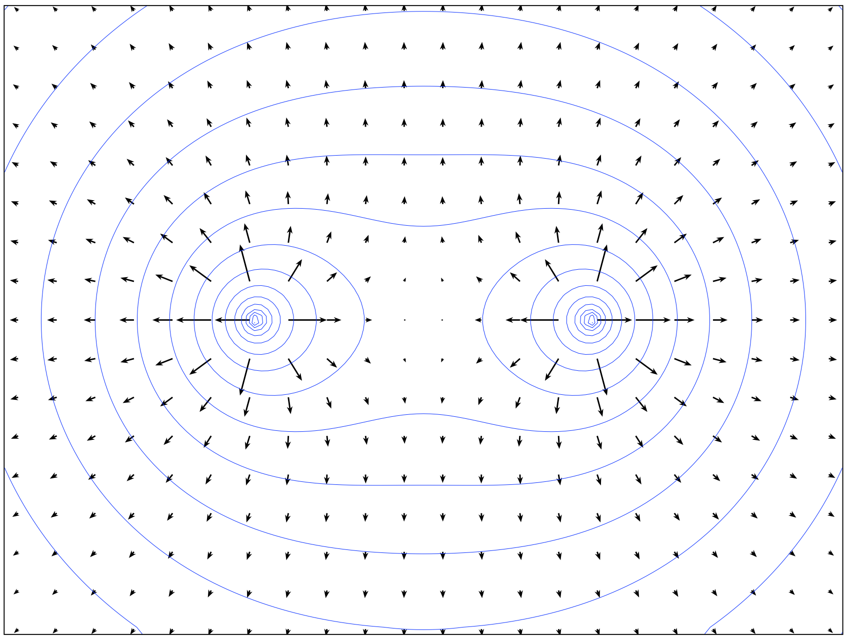


Figure 2.1: Example of two particles with the same charge, the induced force field (arrows) and the potential field (contours)

2.2 The Multipole Framework

This section presents the multipole framework presented in [Hac05]. All lemmata and theorems are taken from [Hac05] and are presented without proof or error bounds; for details see [Hac05] and [GR87].

Lemma 2.2.1 (Potential Field of a Charged Particle). *Given a particle c_i with charge q_i located at point $p_i \in \mathbb{C}$ and an arbitary point $z_0 \in \mathbb{C}$. Then, for any $z \in \mathbb{C}$ outside the circle centered at z_0 with radius $|p_i - z_0|$ the potential energy at point z due to particle c_i is given by*

$$\mathcal{E}_i(z) = q_i \cdot \log(z - p_i) = q_i \left(\log(z - z_0) - \sum_{k=1}^{\infty} \frac{(p_i - z_0)^k}{k \cdot (z - z_0)^k} \right)$$

Theroem 2.2.1 (Multipole Expansion Theorem). *Suppose that m charged particles $c_1, \ldots, c_m$ with charges $q_1, \ldots, q_m$ are located at points $p_1, \ldots, p_m$ inside a circle of radius r with center z_0. Then, for any $z \in \mathbb{C}$ with $|z - z_0| > r$, the potential energy at point z induced by the m charged particles is given by*

$$\mathcal{E}(z) = a_0 \cdot \log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z - z_0)^k} \tag{2.1}$$

where

$$a_0 = \sum_{i=1}^{m} q_i \quad \text{and} \quad a_k = \sum_{i=1}^{m} \frac{-q_i(p_i - z_0)^k}{k}$$

Assume we know the coefficients $a_0, \ldots$, then we get a function which describes the potential field outside the circle and we can evaluate it in time which is dependent on the number of coefficients and not on the number of particles inside the circle. This leads to the idea to truncate the Laurrent series in (2.1) and only use the first p terms, resulting in the so called p-term multipole expansion

$$M_{z_0}^p(z) = a_0 \cdot \log(z - z_0) + \sum_{k=1}^{p} \frac{a_k}{(z - z_0)^k} \tag{2.2}$$

We are now able to approximate the potential field outside a circle containing multiple particles.

As an example consider a set of n points inside a circle with radius r centered at z_0 and we want to evaluate the potential field of these particles at m points outside the circle. The direct evaluation of the potential field function has to be evaluated m times and each evaluation takes time $O(n)$. Therefore the total time needed is $O(n \cdot m)$. Whereas when using a p-term multipole expansion, we need time $O(n \cdot p)$ to compute the $p+1$ coefficients for all n points and evaluating the expansion m times takes $O(m \cdot p)$, which results in a total running time of $O((n + m) \cdot p)$. The next lemma will give us the ability to shift the center of a multipole expansion.

Lemma 2.2.2 (Translation of a Multipole Expansion). *Suppose that $\mathcal{E}(z) = a_0 \cdot \log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z-z_0)^k}$ is a multipole expansion of the potential field due to a set of charged particles that are located inside a circle of radius r and center z_0. Then, for any z outside a circle centered at z_1 with radius $r + |z_0 - z_1|$, the potential field induced by these particles is given by*

$$\mathcal{E}(z) = a_0 \cdot \log(z - z_1) + \sum_{l=1}^{\infty} \frac{b_l}{(z - z_1)^l}$$

where the shifted coefficients are

$$b_l = \frac{-a_0(z_0 - z_1)^l}{l} + \sum_{k=1}^{l} a_k (z_0 - z_1)^{l-k} \binom{l-1}{k-1}$$

Given two p-term multipole expansions $M_{z_1}^p, M_{z_2}^p$ and their coefficients centered at different points z_1, z_2, we are able to translate the centers to a common point and add the coefficients to obtain one multipole expansion approximating the total potential field. Multipole expansions offer a powerful tool for approximating the potential field function outside a circle. However, in the following the concept of local expansions is introduced. In contrast to a multipole expansion, a local expansion is used to approximate the function inside the circle due to particles located outside the circle. We begin with the conversion of a multipole expansion into a local expansion.

Lemma 2.2.3 (Conversion of a Multipole Expansion into a Local Expansion). *Suppose that a set of charged particles is located inside a circle centered at z_0 of radius r and the corresponding multipole expansion is given by $\mathcal{E}(z) = a_0 \cdot \log(z - z_0) + \sum_{k=1}^{\infty} \frac{a_k}{(z-z_0)^k}$. Furthermore, z_1 is a point with $|z_1 - z_0| > 2r$. Then, inside a circle of radius r and center z_1 the potential field is given by the power series*

$$\mathcal{E}(z) = \sum_{l=0}^{\infty} b_l \cdot (z - z_1)^l \tag{2.3}$$

with the converted coefficients

$$b_0 = a_0 \cdot \log(z_1 - z_0) + \sum_{l=1}^{\infty} \frac{a_k}{(z_1 - z_0)^k} \quad \textit{and}$$

$$b_l = \frac{(-1)^{l+1} a_0}{(z_1 - z_0)^l \cdot l} + \frac{1}{(z_0 - z_1)^l} \sum_{k=1}^{\infty} \frac{a_k}{(z_1 - z_0)^k} \binom{l+k-1}{k-1} \quad \textit{for} \quad l > 0$$

Like the multipole expansion the power series in (2.3) is truncated and named p-term local expansion:

$$L_{z_0}^p(z) = \sum_{k=0}^{p} a_k (z - z_0)^k \tag{2.4}$$

We revisit and extend the example from the multipole expansion: Assume we have two p-term multipole expansions centered at z_1 and z_2, each approximating the potential field for n_1 (respectivly n_2) particles, and we want to evaluate the potential field at m points contained in a circle centered at z_0 far enough away. Then, we can convert each multipole expansion to a local expansion centered at z_0. It is easy to see that two local expansions with the same center can be added by adding their coefficients in order to obtain one local expansion describing the potential field inside the circle due to the two sets of particles. The last remaining tool required is the translation of a local expansion.

Lemma 2.2.4 (Translation of a Local Expansion). *Given a local expansion $\mathcal{E}(z) = \sum_{k=0}^{\infty} b_k \cdot (z - z_0)^k$ that describes the potential energy inside a circle centered at z_0 with radius r due to a set of particles that are located outside the circle. Then, for any z inside a circle centered at z_1 with $|z_1 - z_0| < r$ and radius $r' = r - |z_1 - z_0|$, the potential field induced by these particles is given by*

$$\mathcal{E}(z) = \sum_{l=0}^{\infty} b_l (z - z_1)^l,$$

where the translated coefficients are

$$b_l = \sum_{k=l}^{\infty} a_k (z_1 - z_0)^{k-l} \binom{k}{l}.$$

The lemma enables the shifting of the center of a local expansion, but the translation is limited by the circle of the source expansion.

Since we are not interested in the potential field, but in the force vector field, the derivate of the potential field is calculated. For the p-term multipole expansion from Equation 2.2 we obtain

$$\mathcal{E}'(z) = \frac{a_0}{z - z_0} + \sum_{k=1}^{\infty} \frac{k \cdot a_k}{(z - z_o)^{k+1}}$$

and for the p-term local expansion (Equation 2.3)

$$\mathcal{E}'(z) = \sum_{k=1}^{\infty} a_k \cdot k \cdot (z - z_0)^{k-1}$$

Like in [GR87] the force vector field can be obtained by evaluating the gradient at z:

$$\vec{E}(x) = -(\mathrm{Re}(\mathcal{E}'(z)), -\mathrm{Im}(\mathcal{E}'(z))) \quad \text{where} \quad z = (x_1 + x_2 i) \in \mathbb{C}$$

As a result, the multipole method enables the approximation of the force field $\vec{E}_i(x)$. We summarize the most important parts:

- The potential field is the complex function $\mathcal{E}(z)$.
- A multipole expansion is used to describe the potential outside a given circle due to particles inside the circle.
- A local expansion is used to describe the potential inside a given circle due to particles outside the circle.
- Both the multipole and local expansions center can be shifted.
- A p-term expansion is used as an approximation for $\mathcal{E}(z)$.
- The force field can be obtained by using the derivate of a local expansion.

Furthermore the runtime needed for the different operations on a p-term expansion are as follows:

- Computing the p-term multipole coefficients for a point takes time $O(p)$.
- Translating p-term multipole or local expansions takes $O(p^2)$.
- Conversion of a p-term multipole in a p-term local expansion takes $O(p^2)$.
- Evaluating a p-term multipole or local expansion derivate takes time $O(p)$.

For a constant number of coefficients, the running time for each operation is $O(1)$. It remains to find a partitioning of the particles in a way, such that we can apply the multipole method. Most multipole methods, using multipole and local expansions, follow a basic principle consisting of three major steps:

1. For each partition, the multipole coefficients for the particles contained in the partition have to be calculated.
2. Considering one partition, the local coefficients are obtained by finding a set of partitions, which are far enough away in order to apply Lemma 2.2.3.
3. Evaluate for each point the local expansion of its partition(s).

Both [GR87] and [Hac05] use a spatial space decomposition data structure called the quadtree as a partitioning. The next two chapters deal with this problem in detail, and therefore we will first discuss some implementation aspects of the mulitpole framework which are independent of the underlying partitioning.

2.3 Complex Arithmetic with SSE

In order to use Intel's *Streaming SIMD Extensions* to accelerate the computation required when using the Fast Multipole Method, the arithmetic of complex numbers has to be translated into SSE Intrinsics. Therefore we implemented a C++ class, called `ComplexDouble` for complex numbers, which depends on the compiler ability to inline functions. Basically, this class is used as wrapper for SSE and does not store any persistent data for a complex number. Instead, an instance has a reference to a 128-bit register, where the real and imaginary part is stored during a computation.

Listing 2.1 shows the basic structure of the class with the implementation of the constructors, the addition and multiplication operators, and a function to store the value back to memory. The first constructor is used to load two 16-byte aligned double precision values into the 128-bit register, where the lower 64 bit represent the real part. To load the real and imaginary part from standard variables the second constructor can be used. After initializing an instance, it can be used for normal computation. This allows us to write code, which executes SSE intrinsics and still benefits from the comfortable C++'s operator overloading mechanism.

As an example for motivating the usefulness of such a class, Listing 2.2 shows the implementation of the rather simple function `fast_multipole_p2m`, which computes the multipole coefficients for a point charge and adds them to an existing set of coefficients using the multipole expansion theorem. Considering the multiplication of two complex numbers alone, it is obvious that using such a wrapper class results in cleaner and less error-prone code in non trivial computations.

```
class ComplexDouble
{
public:
    __m128d reg;

    inline ComplexDouble(__m128d r)
    {
        reg = r;
    };

    inline ComplexDouble(double* ptr)
    {
        reg = _mm_load_pd(ptr);
    };

    inline ComplexDouble(double x, double y)
    {
        reg = _mm_setr_pd((x), (y));
    };

    inline ComplexDouble operator+(const ComplexDouble& other) const
    {
        return ComplexDouble(_mm_add_pd(reg, other.reg));
    };

    inline void store(double* ptr) const
    {
        _mm_store_pd(ptr, reg);
    };

    inline ComplexDouble operator*(const ComplexDouble& other) const
    {
        __m128d b_t = _mm_shuffle_pd(other.reg, other.reg,
            _MM_SHUFFLE2(0, 1));
        __m128d left = _mm_mul_pd(reg, other.reg);
        __m128d right = _mm_mul_pd(reg, b_t);
        left = _mm_mul_pd(left, _mm_setr_pd(1.0, -1.0));
        return ComplexDouble(_mm_hadd_pd(left, right));
    };

    ...
}
```

Listing 2.1: Parts of the implementation of the wrapper class for complex number arithmetic using SSE Intrinsics.

```
void fast_multipole_p2m(double* coeff,
                        __uint32 numCoeff,
                        float centerX,
                        float centerY,
                        float x,
                        float y,
                        float q)
{
    // a0 += q_i
    coeff[0] += q;
    // a_1..m;
    ComplexDouble ak;
    // p - z0
    ComplexDouble delta(ComplexDouble(x, y) - ComplexDouble(centerX,
        centerY));
    // (p - z0)^k
    ComplexDouble delta_k(delta);
    for (__uint32 k=1; k<numCoeff; k++)
    {
        // load ak from the coeffcient array
        ak.load(coeff+(k<<1));
        // add the coefficient ( -q (p - z0)^k ) / k for this particle
        ak -= delta_k*((double)q/(double)k);
        // store the value back
        ak.store(coeff+(k<<1));
        // next power of (p - z0)
        delta_k *= delta;
    };
};
```

Listing 2.2: Implementation of the function which is used to compute the multipole coefficients for a point charge.

Chapter 3

The Quadtree

First, we will give a brief introduction to quadtrees in general and define some terms for later use. Then we present an algorithm to construct a reduced quadtree for points located on an integer grid. Unlike other approaches, our algorithm can handle very large grids, like for example $2^{32} \times 2^{32}$, to exploit the fact that the bit representation of the integer coordinates can be used for construction. The running time of the algorithm is $O(n \log n)$ for a fixed grid size. The algorithm depends on the distribution of the points, because it is required that only a constant number of points share the same grid position. Otherwise, the algorithm will still produce a valid result and achieve the given runtime, but the overall performance of the force directed algorithm is no longer sub-quadratic.

3.1 Preliminaries

The quadtree belongs to the family of *spatial-space decomposition* data structures and has a wide range of applications. In our application we want to decompose a set of points in the plane. These types of quadtrees are called Point-Region Quadtrees (PR-Quadtrees). The basic idea is to recursively subdividing a square into four equal sized sub-squares until a special condition is met. Since in literature many different terms exist for the same type of quadtree, we will stick to the ones defined [Hac05]. In the following a graph node is referred to as a point to avoid confusing graph nodes and tree nodes. However, we will first introduce a type of quadtree that splits each square until a given depth is reached, regardless of any points.

Definition 3.1.1 (Truncated Quadtree). *The quadtree which results in recursively subdividing a square into four sub-squares until depth D is reached, is called a* truncated quadtree *with depth D.*

It is easy to see that the leaves of such a quadtree form a regular $2^D \times 2^D$ grid. Suppose, we split a square with size 2^D and a set of n points $(x_1, y_1), \ldots, (x_n, y_n)$ is given, then we can assign each point a grid coordinate $(x', y') \in \{0 \dots 2^D - 1\} \times \{0 \dots 2^D - 1\}$ and therefore a leaf in the truncated quadtree. First the bounding box $(x_{\min}, y_{\min}, x_{\max}, y_{\max})$ is calculated by iterating over all points. Then we set $l_{bb} = \max\{x_{\max} - x_{\min}, y_{\max} - y_{\min}\}$ to the longest side length. Finally, we translate and scale each point, so it fits on the grid:

$$\begin{pmatrix} p'_x \\ p'_y \end{pmatrix} = \left\lfloor \begin{pmatrix} p_x - x_{\min} \\ p_y - y_{\min} \end{pmatrix} \cdot \frac{2^D - 1}{l_{bb}} \right\rfloor$$

Obviously, this can be done in time $O(n)$ for all points. Note, when point coordinates are given in floating point representation, this affine transformation is not that simple, because of precision problems due to the fact that floating points do not form a regular grid. In practice, it is sufficient to use a 32-bit regular grid for 32-bit floating points, because in our application all points repulse each other, unlike in other areas like for example gravitational simulations, where particles will form clusters. Since a truncated quadtree of depth D contains 4^D nodes, it is not practical for $D = 32$. Instead another type of quadtree is used, whose node count is linear in the number of points.

Definition 3.1.2 (Reduced Quadtree). *Given a set of n distinct points located on a grid such that $p_i \in \{0, \dots, 2^D - 1\} \times \{0, \dots, 2^D - 1\}$ for a constant $D \in \mathbb{N}$. The quadtree $T = (V, E)$ which results from splitting each node until exactly one point is contained in each leaf, and then replacing each degenerated path $P = \{v_1, \dots, v_k\}$ where the nodes $v_1, \dots, v_{k-1}$ only have one child with an edge (v_1, v_k), is called a* reduced quadtree.

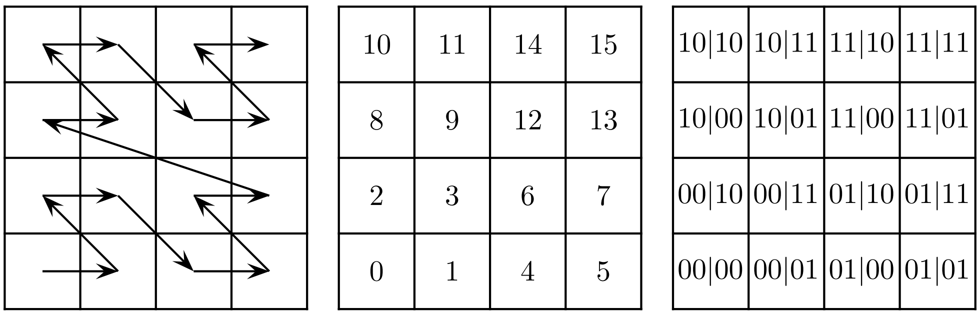
Instead of reproving some important properties of a reduced quadtree, we refer the reader to [Hac05] for details and get:

- Each inner node of a reduced quadtree has at least two children and therefore contains at least two points in its subtree.
- Given n points, the reduced quadtree $T = (V, E)$ contains $\frac{4}{3}n - \frac{1}{3} \leq |V| \leq 2n - 1$ nodes, which is linearly bounded by the number of points.

Furthermore, we can construct a reduced quadtree from a truncated quadtree by deleting recursively all empty subtrees, and then removing all inner nodes with only one child by assigning the child to its grandparent. The result is a quadtree, where all leaves contain at least one point and all inner nodes have at least two children. For later use we define the level of a node.

Definition 3.1.3 (Level of a Node). *Given a node $v \in T$ of a truncated quadtree, the level(v) is defined as the distance from the leaf level. Therefore the root has level $D - 1$ and all leaves have level 0.*

Note that the level of a node in a reduced quadtree is based on the corresponding node of the truncated quadtree. Suppose v is a node in the truncated quadtree, and u is



10	11	14	15
8	9	12	13
2	3	6	7
0	1	4	5

10|10	10|11	11|10	11|11
10|00	10|01	11|00	11|01
00|10	00|11	01|10	01|11
00|00	00|01	01|00	01|01

(a) Morton Order (b) Morton Numbers (c) Binary Representation

Figure 3.1: Morton order on a 4×4 grid (a) and the corresponding decimal morton numbers (b) and binary representation (c).

the parent of v, then obviously level(u) = level(v) + 1, whereas in a reduced quadtree level(u) $\geq$ level(v) + 1 holds. Unlike [Hac05], we do not construct a truncated quadtree as part of the construction process. Instead we compute for each point the path in this tree to construct the reduced quadtree.

Therefore, we introduce the concept of *space filling curves*. Especially the *morton order*, sometimes also referred to as Z-order or *DFS*-order. Figure 3.1 shows the principle of the morton order on a 4×4 grid. Basically a space filling curve describes a principle, how to map a point from the plane (or from higher dimensions) to a one dimensional interval. In our case we map a point on a $2^D \times 2^D$ grid to an integer in the range of $\{0..4^D - 1\}$. We refer to this number as the *morton number* of the point. There exist a lot of other space filling curves. However, we use the morton order, because the morton number can be easily calculated by alternately taking the bits of x and y from the most significant to the least significant bit [BET93].

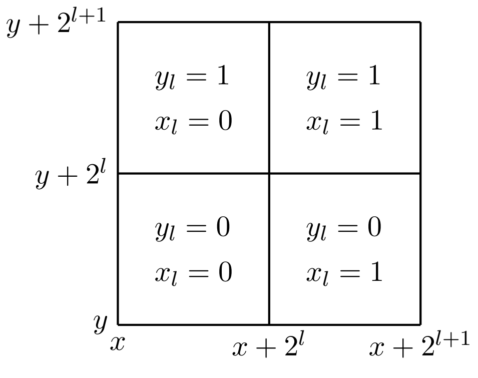


Figure 3.2: A subdivision of a quadtree cell at level $l + 1$ with coordinates (c_x, c_y) at the bottom left and size 2^{l+1}; x_i and y_i denote the i-th bit of x and y respectively, where x_1 is the least significant bit.

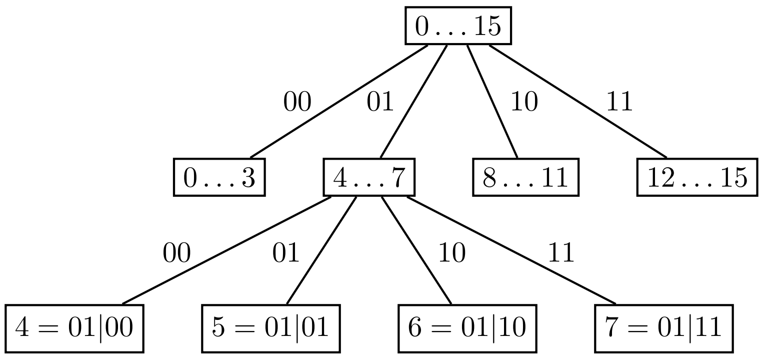


Figure 3.3: The path from the root to the leaf in the truncated quadtree described by the morton numbers 4..7. For each level two bits are required.

Definition 3.1.4 (Calculating the Morton Number). *Let p be a point with D-bit integer coordinates $(x, y) \in \{0 \dots 2^D - 1\} \times \{0 \dots 2^D - 1\}$ and let x_i be the i-th bit of x (analogue y_i). Then the morton number m with $2D$ bits is calculated by interleaving the bits of x and y:*

$$m := y_D, x_D, \dots, y_1, x_1$$

Like in [Gar82] and [BET93], the morton number describes for each grid cell the path from the root to the leaf in our truncated tree with exactly two bits for each level of the tree (see Figure 3.2).

Suppose we have given a point set, then we can calculate for each point the morton number in time $O(D)$. We can sort the points by their morton numbers in time $O(n \log n)$ (assuming comparing two morton numbers takes constant time). Then the points form the same sequence as if they would be discovered in an inorder traversal of the tree. As stated above the morton number describes the path in the truncated quadtree. Suppose, we have given two points p, q and their morton numbers, comparing the bits and determining the index of the most significant bit that differs enables the computation of the level of their common ancestors in the truncated quadtree. For example, assume that the 5 most significant bits are equal and the sixth bit differs, then the first $\lfloor 5/2 \rfloor = 2$ nodes (excluding the root) are common ancestors of p and q (see Figure 3.4). We define the lowest common ancestor level of p and q as follows:

Definition 3.1.5 (Lowest Common Ancestor (LCA)). *Given two points p and q on a $2^D \times 2^D$ grid. The root of the smallest subtree containing p and q is called the* lowest

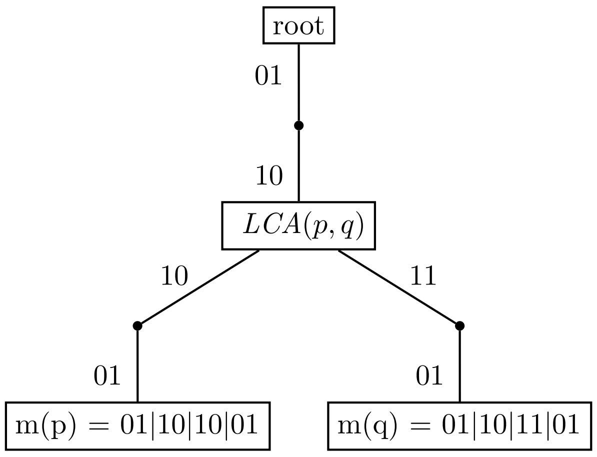


Figure 3.4: Example for two points $p = (9, 6)$ and $q = (11, 6)$ and their morton numbers (105 and 109) on a 16×16 grid

common ancestor $LCA(p, q)$. *The level of this node is referred to as* $LCAL(p, q)$. *By calculating the morton numbers* $m(p)$ *and* $m(q)$ *of* p *and* q*, LCAL(p,q) can be obtained by*

$$LCAL(p,q) = \left\lfloor \frac{\mathrm{msb}(m(p) \otimes m(q))}{2} \right\rfloor + 1,$$

where msb *denotes the index of the most significant bit and* $\otimes$ *the bitwise XOR.*

The bitwise XOR of the two morton numbers sets all bits that differ to 1 and all equal bits to 0. Therefore, the index of the most significant bit of this sequence is the first place where the two morton numbers differ. This needs to be divided by two because we have two bits for each level, and incremented by one to receive the index of the first equal 2-bit block.

3.2 Quadtree Construction

Now, we have all tools to construct a reduced quadtree from the information provided by $LCAL$. Listing 3.1 shows the main algorithm. First, for each point the morton number m_i is computed, and then the points are sorted by ascending morton numbers. Since the morton order defines the order of the leaves in the truncated quadtree, it also defines the order in the reduced quadtree. The basic idea is to iterate over all points and to construct the tree bottom-up (see Algorithm 3.2). Suppose three consecutive points p_{k-1}, p_k, p_{k+1} are given and we have already constructed a valid reduced quadtree for p_1 to p_{k-1}. Two cases can arise when comparing $LCA(p_{k-1}, p_k)$ and $LCAL(p_k, p_{k+1})$.

Algorithm 3.1 Main Quadtree Construction Function

Input: point sequence $P = \{p_1, \ldots, p_k\}$

Global: point index k

Output: root of reduced quadtree

```
1: function buildTree
2:     for $i \leftarrow 1, .., n$ do
3:         $m[i] \leftarrow$ computeMortonNr($p_i$)
4:     end for
5:     sortPointsByMortonNr($P$)
6:     $k \leftarrow 1$
7:     return buildTreeUntil($LCAL(1, n) + 1$)
8: end function
```

Algorithm 3.2 Recursive Bottom-Up Construction

Input: point sequence $p_1, \ldots, p_k$ sorted by morton number

Global: current point index k

Output: root v of subtree with level < maxLevel

```
1: function buildTreeUntil($maxLevel$)
2:     $v \leftarrow$ newLeaf($p_k$)
3:     $k \leftarrow k + 1$
4:     while $k \leq n \wedge LCAL(p_{k-1}, p_k) < maxLevel$ do
5:         $L_{\text{left}} = LCAL(p_{k-1}, p_k)$
6:         $L_{\text{right}} = LCAL(p_k, p_{k+1})$
7:         if $L_{\text{left}} > L_{\text{right}}$ then
8:             $w \leftarrow$buildTreeUntil($L_{left}$) // Case 2
9:         else
10:            $w \leftarrow$ newLeaf($p_k$) // Case 1
11:            $k \leftarrow k + 1$
12:        end if
13:        $v \leftarrow$merge($v, w, L_{left}$)
14:    end while
15:    return $v$
16: end function
```

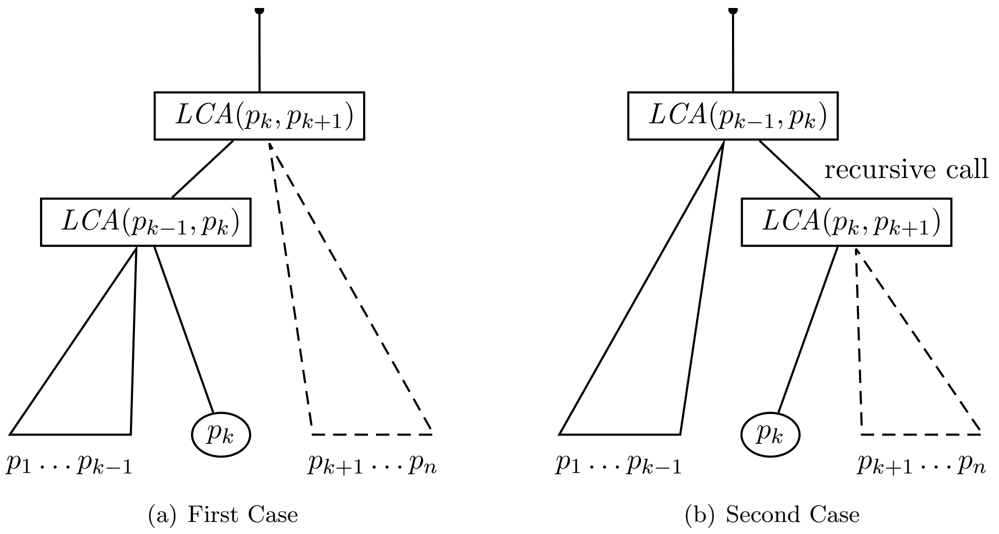


Figure 3.5: The two cases for point p_k: (a) the point can be merged in our current subtree, (b) the right subtree has to be constructed first by a recursive call before merging it.

In the first case (Figure 3.5(a)), when $LCAL(p_{k-1}, p_k) \leq LCAL(p_k, p_{k+1})$, then the node $LCA(p_{k-1}, p_k)$ can be created before $LCA(p_k, p_{k+1})$. Hence p_k can be merged into the current subtree by either creating a new common ancestor for the current root v and the leaf containing p_k or appending the leaf to v (Algorithm 3.3). Note that the parent of p_k cannot be in the already constructed quadtree on a lower level than the current root. Otherwise, it becomes clear that this potential parent has to be a common ancestor of p_{k-1} and p_k, because the points are sorted, and the current root is a common ancestor of p_{k-1} and one of its predecessors. However, the algorithm has visited p_{k-1} before and then the second case would have been applied.

In the second case $LCAL(p_{k-1}, p_k) > LCAL(p_k, p_{k+1})$ holds (see Figure 3.5(b)), the subtree containing p_k and p_{k+1} has to be constructed first, since their common ancestor is on a lower level than the common ancestor of the current root and the leaf of p_k. We start a recursive call to construct the subtree starting with p_k until the tree is not higher than our current root. Then we can merge the root of this subtree with our current root by calling MERGE to check if the result of the recursive call is either a sibling or a child of our current root.

Lemma 3.2.1. *Given n distinct points on a $\{0, \dots, 2^D - 1\} \times \{0, \dots, 2^D - 1\}$ grid with fixed D, then Algorithm 3.1 constructs a reduced quadtree in time $O(n \log n)$.*

Proof. As stated above the interleaving of bits for calculating the morton number of two integer coordinates can be done in constant time for a fixed bitlength. Sorting takes $O(n \log n)$ time. It remains to show the time needed by the recursive construction. First, the subroutine MERGE takes $O(1)$. Note, each recursive call to BuiltTreeUntil creates at least one leaf and increments the point index k by one. It follows that in each iteration

Algorithm 3.3 Recursive Bottom-Up Construction Merge

Input: Two nodes a, b which are either siblings or a is the parent of b, level of their common ancestor

Output: Common ancestor of a and b

```
1: function MERGE($a, b, LCAL_{ab}$)
2:     if $LCAL_{ab} = \text{level}(a)$ then
3:         $a.appendChild(b)$
4:         return $a$
5:     else
6:         $c \leftarrow$ NEWNODEWITHCHILDREN($a, b$)
7:         $\text{level}(c) \leftarrow LCAL_{ab}$
8:         return $c$
9:     end if
10: end function
```

inside the while loop, k is incremented. Since there are only n points, the running time for BUILDTREEUNTIL is linear. □

Since the level of a reduced quadtree node v, referred to as level(v), is based on the level in the truncated quadtree, it can be used to determine the size of the cell and, in combination with the morton number of a point contained in v, to compute the center of the cell. Given a node v, the size is obtained by

$$\text{size}(v) = 2^{\text{level}(v)} \cdot \frac{l_{bb}}{2^D - 1}.$$

We can calculate the center of a quadtree cell from the level and the morton number of any point inside the cell by taking the first $2 \cdot \text{level}(v)$ bits (which are the same for all points inside the cell), and reverse the interleaving of bits. As a result we get the bottom-left coordinate (x', y') of the cell on the grid. Hence, the center is computed by transforming the grid coordinates and adding half of the previously computed side length:

$$\text{center}(v) = \begin{pmatrix} x' \\ y' \end{pmatrix} \cdot \frac{l_{bb}}{2^D - 1} + \begin{pmatrix} \text{size}(v) \\ \text{size}(v) \end{pmatrix} \cdot \frac{1}{2}$$

The presented method results in a rather short implementation. However, the algorithm has a few disadvantages. The computational cost of both the computation of LCAL and the tree construction functions are very expensive (in relation to the remaining functionality used inside the recursive function). When it comes to parallel execution of code, the recursive function is a problem. On the one hand the algorithm performs good in practice because it does not need to read data ahead. We can start a recursive call without knowing the index of the last point in the subtree. On the other hand this leads to the problem of estimating the amount of work which will be done by a recursive call.

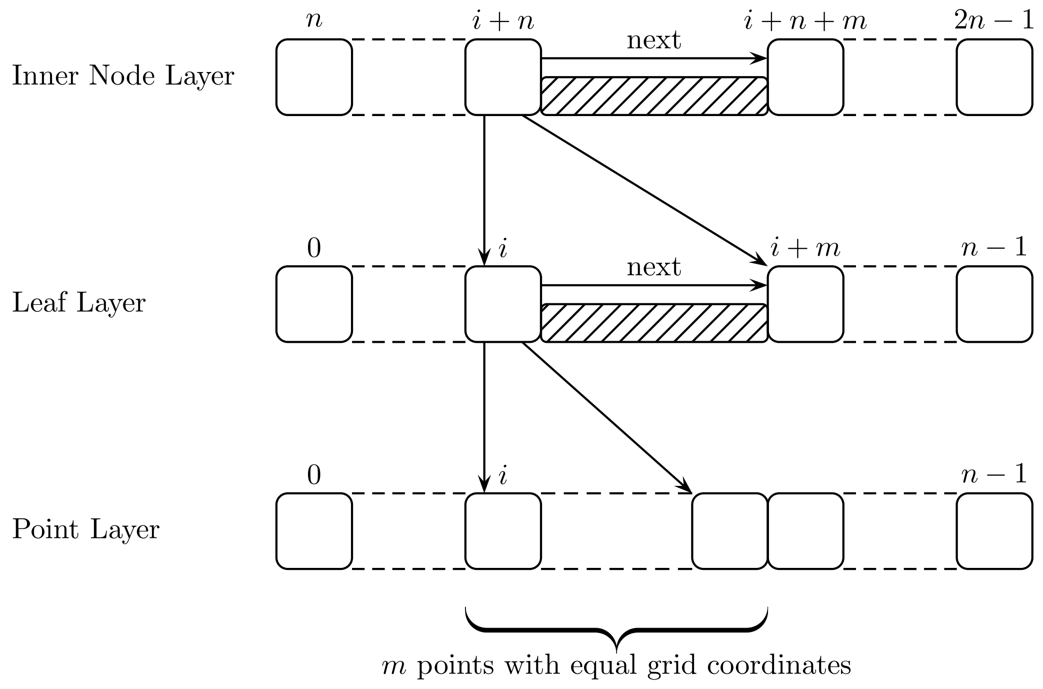
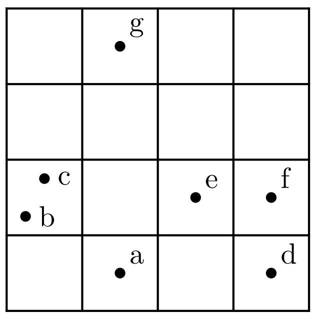


Figure 3.6: Initial memory layout for the quadtree construction.

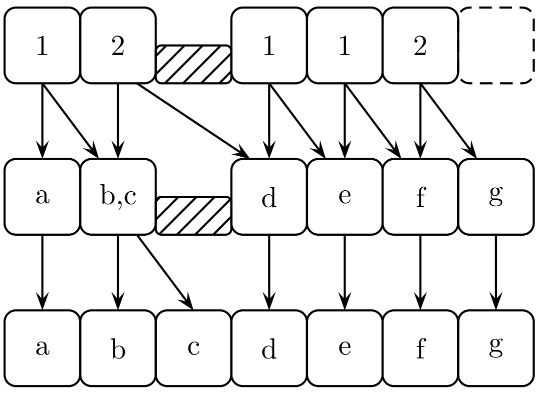
This leads to the idea of preparing the tree in parallel as far as possible. Then a similar recursive algorithm is used to just link the prepared nodes to each other which will result in the desired tree. Like in the previous algorithm, we assume that n points are given, which are sorted by their morton numbers. In the following we refer to the sorted point sequence as *point layer*. For the tree structure, memory for $2n$ nodes is allocated. Each node record consists of the following attributes:

- level: the level of the node
- next: the index of the next node
- children: a list or array of children

Note that $2n-1$ nodes would be enough, but for simplicity and a feature which is not needed here but useful for other applications, one more is allocated. The leaves are located in the first n entries of the array; we refer to this part as the *leaf layer*. The remaining last n entries are used for the inner nodes and referred to as the *inner node layer*. Since in our quadtree each leaf corresponds to a grid cell and not to a point we still have to deal with the problem of handling the special case where more than one point is contained in a cell. If two points are located in the same cell, they have equal grid coordinates. Thus, the resulting morton numbers have to be the same. Since the points are sorted by their morton numbers, all points sharing the same cell form a continuous sequence in the



(a) Example Distribution



(b) Initial Memory

Figure 3.7: Example for a point distribution and the resulting prepared tree: (a) seven points $a, \ldots, g$ located in grid cells, (b) the corresponding prepared tree and its memory layout.

point layer. For each occurring morton number in the point layer, a leaf is created in the leaf layer at position i of the first point in this sequence. The leaf is linked via the next reference to its succeeding leaf at position $i + m$.

On the inner node layer a node is placed at position $i + m + n$ and the two leaves at position i and $i + m$ are assigned to the node as children. The level of the inner node is set to the $LCAL(p_i, p_{i+m})$. Basically, an inner node is created as lowest common ancestor for two adjacent leaves. Like the leaves they are linked by a next reference to form a chain.

Figure 3.6 illustrates the procedure for each layer and the space wasted by this approach. In Figure 3.7 an example point distribution and the resulting memory of prepared tree is displayed. The question arising is, why wasting this amount of space? In this application the quadtree has to be built several times for the same amount of points but with different coordinates. Thus, it is useful to preallocate memory for $2n$ nodes to avoid the use of expensive memory allocation during the tree construction process. Furthermore, the positions of the leaves and inner nodes in the above described procedure depend only on the point layer and not on any preceding leaves or inner nodes. This is very useful in order to prepare the tree in parallel. The points are evenly distributed among the threads, so an interval is assigned to each thread which the thread can then prepare. The unused array entries between two leaves and two inner nodes are caused by the number of points in one grid cell. In this application a large grid is used, thus, in practice the number of points sharing one grid cell is very small. However, in other applications this might not be the case, therefore we outline the basic idea. The gaps on the two layers induced by m points sharing the same grid cell can be used to construct a tree for the m points recursively. Each gap consists of $m - 1$ unused entries plus the leaf (or respectively the inner node) resulting in a total of m available entries on each layer for m points. Like in the bigger problem where n array entries on each layer are available for n points, we have

Algorithm 3.4 Recursive Linking of the Tree

```
function LINKTREE($v, level_{\max}$)
    $w \leftarrow \text{next}(v)$
    while $(w \neq 2n-1) \wedge (\text{level}(w) < level_{\max})$ do
        case $\text{level}(v) < \text{level}(w)$ :
            $\text{firstChild}(w) \leftarrow v$
            $v \leftarrow w$
        case $\text{level}(v) = \text{level}(w)$ :
            append all children of $w$ to $v$, except the first
            $\text{next}(v) \leftarrow \text{next}(w)$
        case $\text{level}(v) > \text{level}(w)$ :
            $w \leftarrow$ LINKTREE($w, \text{next}(v)$)
            $\text{lastChild}(v) \leftarrow w$
            $\text{next}(v) \leftarrow \text{next}(w)$
        end cases
        $w \leftarrow \text{next}(v)$
    end while
    return $v$
end function
```

m entries on each layer for m points. Now the complete procedure of sorting the points and preparing the tree is repeated for the points $p_i, \dots, p_{i+m-1}$, consisting of the following steps:

1. Instead of calculating the bounding box of the m points, the size and position of the leaf containing the points is used.
2. Recompute the morton numbers using the new bounding box.
3. Sort the points $p_i, \dots, p_{i+m-1}$ by their new morton numbers.
4. Prepare the tree inside the existing array.

However, in the following this version is not used and we assume the number of points in a grid cell is sufficiently small. Instead we focus on Algorithm 3.4 which recursively adjusts the links of the inner nodes so that we obtain the quadtree. Like in the first version, the basic idea is to iterate from the left to the right, but instead of iterating over the point layer, we follow the inner node chain and adjust the children. To achieve this, the level of the current node v is compared with the level of its successor w in the chain. Furthermore, the following invariant holds: At the end of each iteration, the last leaf in the subtree rooted at v equals the first leaf of the subtree rooted at w. Obviously the invariant holds in the beginning for all inner nodes, because each inner node shares a leaf with its successor. When comparing level(v) and level(w), three cases can arise.

1. The current node v is on a lower level than the succeeding node w. Thus, v becomes the first child of w, and w becomes the new current node. Since the algorithm advances in the chain, v will never be visited again.

2. The current node v is on the same level as w and has to be merged (this merge is similar to the one in the first algorithm). All children of w, except for the first, are added to v, then w is removed from the chain by linking v to the successor of w.

3. The current node v is on a higher level than w. Since w may not be the root of this subtree which will become the last child of v, we have to construct the complete subtree first in order to obtain the root, which is then linked to v. Therefore, a recursive call is made with the condition to construct the subtree starting with w which is not higher than v. When the recursion terminates, it returns the root of this subtree, which replaces the last child of v. Furthermore, the successor of v is set to the successor of the returned root, because all elements between v and the root have been processed.

In the following, an example is given to clarify the algorithm. Figures 3.8 - 3.14 show a detailed example for the seven points and their distribution shown earlier in Figure 3.7.

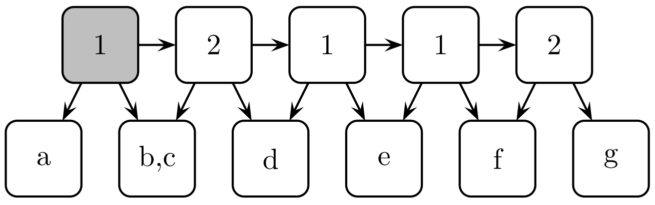


Figure 3.8: The initial tree structure with the prepared leaf and inner node layer. The point layer is not shown for reasons of clarity. The algorithm starts at the first inner node (highlighted in gray). For this node and its successor the first case applies. Thus, the node becomes the first child of its successor and the algorithm advances in the chain.

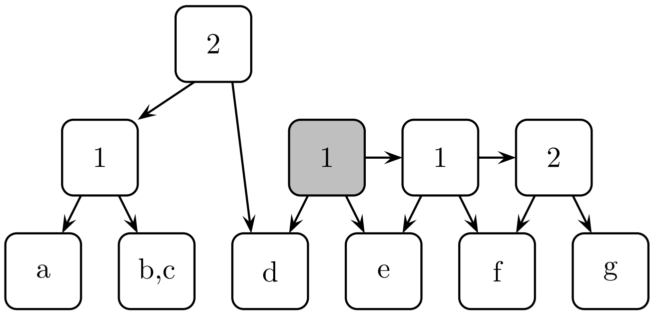


Figure 3.10: In the first step of the recursive call the second case applies, because the succeeding node is on the same level. The list of children of the current node is augmented by the children of the next node. The successor is then removed from the chain by relinking the current node to the target of the successor's next pointer.

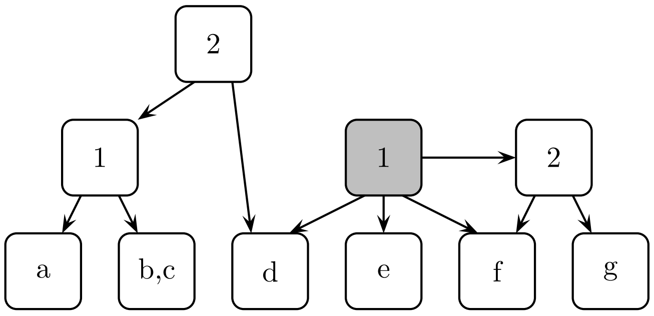


Figure 3.11: The level of the successor is no longer lower than the level of the node in the chain from where the recursive call was initiated. The recursion terminates and returns back to the node which is either on the same level or on a lower level, then the root of the subtree is assigned as last child to the higher node. Each time the recursion terminates, the next link is handed to the higher level.

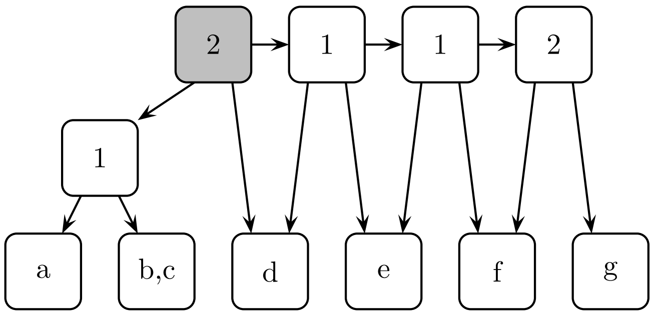


Figure 3.9: Second Step of the example: The next node in the chain is on a lower level than the current one, therefore the third case applies. A recursive call is initiated to build a subtree from the chain until a node is found which is at least on the level of the current node.

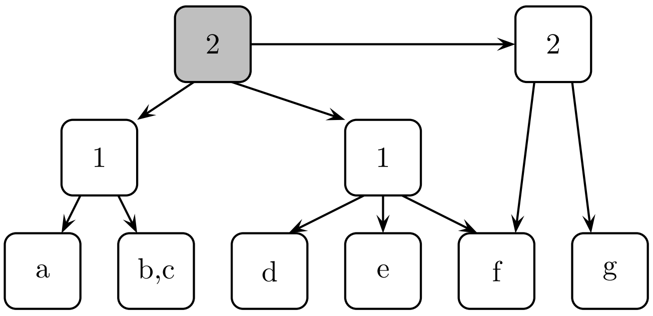


Figure 3.12: The next node is on the same level as our current root. Like before, the second case applies where the lists of children are merged, and the successor is removed from the chain.

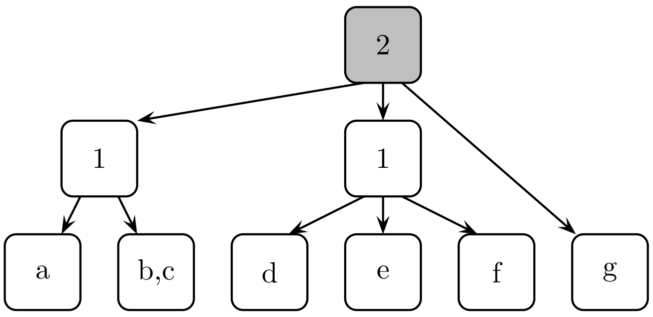


Figure 3.13: The last step of the algorithm. The algorithm terminates and returns the root of the tree as result, since the current node has no longer a valid successor in the chain. The resulting tree is the final quadtree.

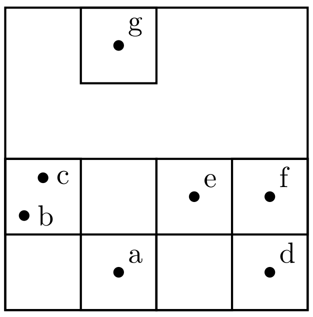


Figure 3.14: The resulting subdivision induced by the quadtree.

Chapter 4

The Well-Separated Pair Decomposition

In order to approximate forces between quadtree cells, these cells must have a minimum distance. Like in [Hac05] we call such two cells *well-separated.* The basic idea is to find for each cell a set of well separated cells. Then we can apply the fast multipole method in order to approximate long distance repulsive forces for a set of points.

4.1 Previous Work

Most other approaches like [Hac05], [APG94] and [GR87] use a so called interaction list, which holds for each node a set of well-separated nodes. These nodes are found by a special tree traversal. Instead of using an interaction list for each quadtree node, we reduce the problem of finding these cells to a problem in computational geometry named the *well-separated pair decomposition (WSPD).* The problem consists of decomposing a set of n points into a set of pairs, where each pair represents two sets of points which are well-separated. Callahan and Kosaraju [Cal95, CK95] have shown that this can be done in time $O(n \log n)$ and the number of pairs is $O(n)$ using a *fair-split tree.* Our approach is very similar to [CK95], where it is already mentioned that a well-separated pair decomposition can be used for the Fast Multipole Method by [GR87]. However, we will not use the fair-split tree, because it depends heavily on linked lists. Instead, the reduced quadtree is used and the presented algorithm is adapted for finding a well-separated pair decomposition.

4.2 Preliminaries

First we define the term well-separated for two point sets like in [Cal95]; see Figure 4.1 for an example.

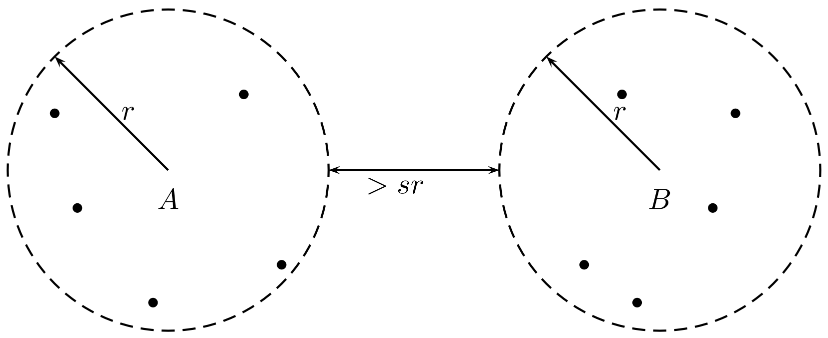


Figure 4.1: Example of two well-separated point sets A and B.

Definition 4.2.1 (Well-Separated). *Given two point sets A, B, we call A and B well-separated if they can be contained in circles with radius r whose distance to each other is more than $s \cdot r$, where $s > 0$ is the separation factor.*

The basic idea is, when we managed to find such a pair (A, B), then we can recursively handle the two sets separately. Consider two subsets $A' \subset A$ and $B' \subset B$. If A and B are well-separated, then obviously A' and B' are well-separated too by choosing the same circles. The goal is to find a set of pairs which covers all points, meaning when we consider any two points then there exists a pair in the decomposition which implies that these two points are well-separated. Note that two points are well-separated by defintion since they can be enclosed in two infinitely small circles. We formalize this in the following definition taken from [CK95]

Definition 4.2.2 (Well-Separated Pair Decomposition). *A well-separated pair decomposition W of a point set P is a a set of pairs*

$$W = \{(A_1, B_1), \dots, (A_n, B_n)\}$$

such that

- $\forall i \in 1, \dots, n : A_i, B_i \subset P$;
- $\forall i \in 1, \dots, n : A_i \cap B_i = \emptyset$;
- *A_i and B_i are well-separated;*
- *for each pair of points $(p, q) \in P \times P$ there exists a pair (A_i, B_i) with $p \in A_i$ and $q \in B_i$ or $p \in B_i$ and $q \in A_i$.*

In the following, the nodes of the reduced quadtree are used as point sets, resulting in the problem of finding a decomposition where $W \subset (V \times V)$. To be more concrete, we define the term well-seperated for two quadtree cells.

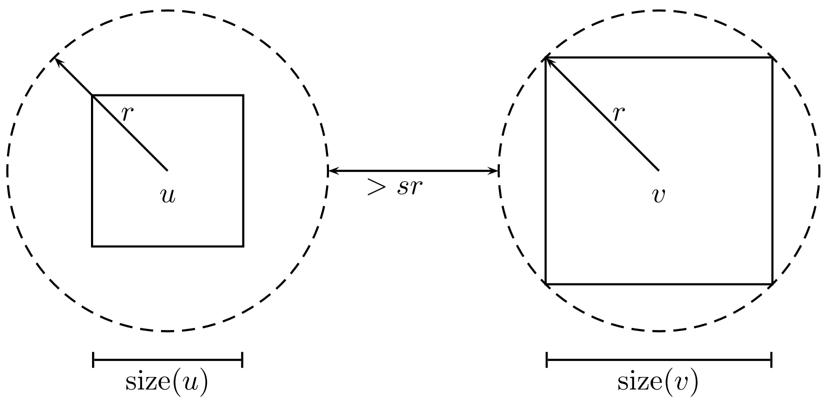


Figure 4.2: Two well separated quadtree cells u and v. The radius r of the enclosing circle is for both cells the maximum.

Definition 4.2.3 (Well-Separated Quadtree Cells). *Given a pair (u, v) of two quadtree nodes and a separation factor $s > 0$. If u and v only contain one point they are well-separated. Otherwise let $d(u, v)$ denote the distance between the centers of u and v and $l_{\max} = \max\{\text{size}(u), \text{size}(v)\}$. The pair (u, v) is called well-separated when the following condition holds:*

$$d(u, v) > (s + 2) \cdot \frac{l_{\max}}{\sqrt{2}}$$

Furthermore, if the pair (u, v) is well-separated, we say u is well-separated from v and vice versa.

Considering Figure 4.2, we have

$$r = \sqrt{2 \cdot \left(\frac{l_{\max}}{2}\right)^2} = \frac{l_{\max}}{\sqrt{2}}.$$

Since the two enclosing circles with radius r should have a distance greater than sr

$$\begin{aligned} d(u, v) - 2r &> sr \\ d(u, v) &> (s + 2)\frac{l_{\max}}{\sqrt{2}}. \end{aligned}$$

Note, instead of using for each cell its own enclosing circle induced by the size of the node, we use for the smaller cell the circle of the bigger one. The reason for this is a condition induced by Lemma 2.2.3 from Chapter 2. Given a pair of two well-separated cells (u, v), our goal is to do a symmetric approximation, meaning we want to calculate the force for all points contained in u induced by the points in v and vice versa. Since the condition for using Lemma 2.2.3 to convert a p-term multipole expansion into a p-term local expansion is not symmetric, we need to ensure that it holds both ways.

Algorithm 4.1 Recursive Well-Separated Pair Decomposition

```
1:  function COMPUTEWSPD($u, v$)
2:      if size($u$) > size($v$) then
3:          swap($u, v$)
4:      end if
5:      if wellSeparated($u, v$) then
6:          $W \leftarrow W \cup \{(u, v)\}$
7:      else
8:          for all $w \in$ children($v$) do
9:              COMPUTEWSPD($u, w$)
10:         end for
11:     end if
12: end function
```

The basic idea of a well-separated pair decomposition is to find a set of pairs of nodes that covers the complete quadtree. Considering only one node v, the goal is to find all nodes that are well-separated from v but not well-separated from the parent of v.

In [CK95] it has been shown that the size of such a decomposition is $O(n)$ using a fairsplit tree. However, the same principle can be applied to quadtrees, because the proof in [CK95] relies on a packing argument which can also be used for a quadtree. Before we prove the size of the decomposition we present the algorithm to calculate such a decomposition.

4.3 Algorithm

Given a point set and the corresponding reduced quadtree, we can now compute the WSPD. We use the same algorithm like in [CK95] with some minor modifications for a quadtree.

The recursive function COMPUTEWSPD (Algorithm 4.1) works as follows: Suppose we have given two distinct subtrees rooted at u and v. If the pair (u, v) is well-separated

Algorithm 4.2 Well-Separated Pair Decomposition for a Reduced Quadtree

```
1: function QUADTREEWSPD(T)
2:     for all $u \in$ innerNodes($T$) do
3:         for all $(v, w) \in$ orderedPairsOfChildren($u$) do
4:             COMPUTEWSPD($v, w$)
5:         end for
6:     end for
7: end function
```

then we add it to our set of pairs. Otherwise assume v is the larger cell and its children are $\{v_1, \dots, v_n\}$. Then we start recursive calls for the pairs $(u, v_1), \dots, (u, v_n)$. In order to compute a WSPD for a complete quadtree (Algorithm 4.2), we try for each inner node to "separate the node from itself". This is achieved by calling the recursive algorithm on all ordered pairs of its children.

Lemma 4.3.1. *Algorithm 4.2 computes a well-separated pair decomposition W for the reduced quadtree T.*

Proof. It is easy to see that each pair which is added to W is well-separated. It remains to show that for a pair $(u, v) \in W$ the two point sets covered by u and v do not overlap. Furthermore each pair of points has to be covered by one pair in W.

First, we observe that a call to Algorithm 4.1 is only done for two nodes (v, w) where v and w are siblings. Algorithm 4.1 may result in some recursive calls but still in distinct subtrees. Therefore a pair $(u, v) \in W$ generated by Algorithm 4.1 satisfies the condition $A_i \cap B_i = \emptyset$

Since we start Algorithm 4.1 on all ordered pairs of children for each internal node of T, the call terminates only when a complete decomposition for the children is generated. Consider the lowest common ancestor v of two points. Algorithm 4.2 tries to separate the children from each other and, because the two points are contained in two different children (otherwise v is not their LCA), the call will generate a pair covering the two points. □

Consider a well-separated pair decomposition W for n points. Obviously $|W|$ can be bounded by $O(n^2)$. But like mentioned above the size of W can be bounded by $O(n)$. In [CK95] this bound has been proven for the fair-split tree and in [HP08] it is done for quadtrees as well. However, we reprove this result by following the basic ideas used there, because we use a slightly different definition of well-separated.

Lemma 4.3.2. *The number of pairs generated by Algorithm 4.2 is $O(n)$ where n is the number of points.*

Proof. Consider a recursive sequence of calls to COMPUTEWSPD, which results in a well-separated pair (u, v). Without loss of generality let this sequence of calls be

1. COMPUTEWSPD(u, v'), not well-separated, split v'
2. COMPUTEWSPD(u, v), well-separated

where v' is the parent of v and u' the parent of u, respectively; see Figure 4.3(a). We will charge the pair $(u, v) \in W$ to v' and claim that v' is only charged constant times. Let v' be fixed. From the sequence of calls we observe:

$$\text{size}(u') \geq \text{size}(v') \geq \text{size}(u)$$

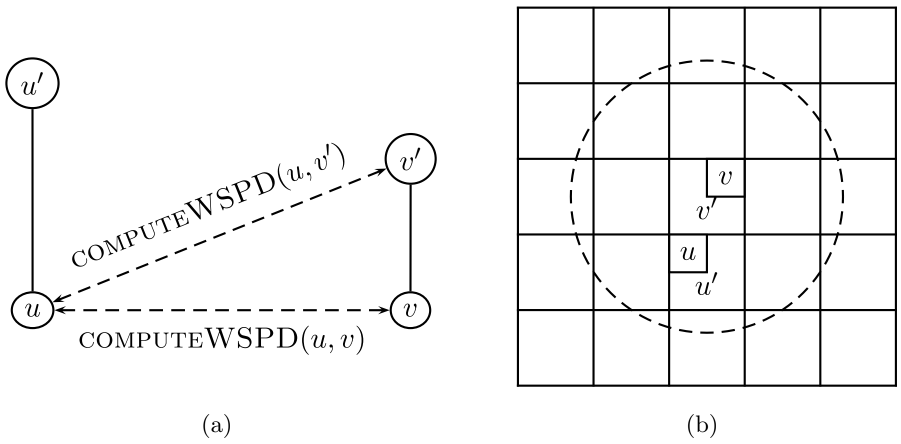


Figure 4.3: The sequence of two calls leading to a well-separated pair (u, v) (a) and the circle around v' overlapping all not well-separated cells (b).

The first call implies that u' has been split before v'. Therefore, u' is at least the size of v', because the algorithm always splits the bigger node. Then we split v' instead of u with the same argument. Now it remains to show that v' is charged only constant times. To achieve this we show that the number of pairs (u', v') is constant for a fixed v'. From the fact that (u', v') is not well-separated, it follows that

$$d(u', v') \leq (s + 2)\frac{\text{size}(u')}{\sqrt{2}}.$$

We can conclude:

- all u' are at least the size of v'
- all u' have to be sufficiently near to v' to be not well-separated

From these two facts it is easy to see that, if we want to maximize the number of u', they have to be as small as possible. Hence we assume they have the same size as v'. This reduces the problem to bounding the number of cells with side length size(v') that overlap a circle of radius $(s + 2)\frac{\text{size}(v')}{\sqrt{2}}$; see Figure 4.3(b). Instead of proving that this number is constant, we refer the reader to [CK95] for details and only outline a simple idea. Suppose, we rescale the problem so that size$(v') = 1$, then the circle has a constant radius of $(s+2)\frac{1}{\sqrt{2}}$, assuming s is fixed. Therefore it can only intersect a constant number of unit cells.

Thus, there are only a constant number of u' and only a constant number of pairs (u', v') for a fixed v'. Since both nodes have at most 4 children, the number of pairs (u, v) is constant and v' is only charged constant times. Furthermore, the reduced quadtree has $O(n)$ nodes and the lemma follows. $\square$

Note that the above lemma delivers a bound for the size of the entire decomposition and not for the number of nodes being well-separated from a single node. However, the result can be used to bound the runtime of the algorithm:

Lemma 4.3.3. *Given a reduced quadtree, Algorithm 4.2 computes a well-sparated pair decomposition in time $O(n)$.*

Proof. Consider the calls to COMPUTEWSPD(u, v) as tree structure, then the main function creates a forest of calls to COMPUTEWSPD(u, v). A call might result in recursive calls, creating a tree of calls. The recursion terminates when a well-separated pair is added to the set. Each call not resulting in a pair generates at least two calls because all inner nodes in the reduced quadtree have at least two children. Therefore, the forest of recursive calls has $O(n)$ leaves (a leaf corresponds to a well-separated pair) and each inner node has at least two children. Thus, the total number of inner nodes is bounded by $O(n)$, and the time bound follows. $\square$

The algorithm described above delivers a set of pairs of quadtree nodes which are well-separated for a given separation s. The basic idea is to apply the fast multipole theorems and lemmata from Chapter 2 (especially Lemma 2.2.3) in order to approximate forces between the two cells of a pair. Suppose the multipole coefficients are given for all quadtree cells and a well-separated pair decomposition W for a separation factor $s = \epsilon$. Then the local coefficients can be obtained by iterating over all pairs and for each pair $(u, v) \in W$ converting the multipole coefficients of u to local coefficients of v using Lemma 2.2.3 and vice versa.

4.4 Improving the Algorithm

Algorithm 4.2 computes a well-separated pair decomposition with a size of $O(n)$. However, using this decomposition in practice would result in a very slow algorithm. The reason for this is the large constants hidden in the bound of the size and the costs for approximating forces between two sets of points.

Consider a pair of distinct cells (u, v). If the number of covered points of both u and v is relatively small, it is cheaper to calculate the forces directly instead of finding a WSPD and approximating them using the Fast Multipole Method. Note, the direct computation does not require a pair to be well-separated. Therefore we can stop the recursion when the number of points contained in the two cells is small enough. Suppose we only want to approximate forces when both point sets have at least C_{pairs} points. Then we can modify the WSPD algorithm in a way so it skips

- a subtree if it has less than C_{nodes} points; and
- a pair if both cells contain less than C_{pairs} points.

When we skip a node or a pair, we need to evaluate the forces directly; therefore we maintain two additional sets D_{nodes} and D_{pairs} for this purpose. The modified versions are Algorithm 4.3 and 4.4. The values of C_{nodes} and C_{pairs} are heavily implementation dependent and can only be found by experiments. In our implementation, we set $C_{\text{nodes}} = 25$ and $C_{\text{pairs}} = 16$. However, the bounds for both runtime and size of the original algorithm remain valid, because this version terminates only earlier resulting in a chopped forest of calls.

Algorithm 4.3 Recursive Bounded Well-Separated Pair Decomposition

```
1:  function COMPUTEWSPD($u, v, C_{\text{pairs}}$)
2:      if numPoints($u$) $\le C_{\text{pairs}} \wedge$ numPoints($v$) $\le C_{\text{pairs}}$ then
3:          $D_{\text{pairs}} \leftarrow D_{\text{pairs}} \cup \{(u, v)\}$
4:      else
5:          if size($u$) $>$ size($v$) then
6:              swap($u, v$)
7:          end if
8:          if wellSeparated($u, v$) then
9:              $W \leftarrow W \cup \{(u, v)\}$
10:         else
11:             for all $w \in$ children($v$) do
12:                 COMPUTEWSPD($u, w, C_{\text{pairs}}$)
13:             end for
14:         end if
15:     end if
16: end function
```

Algorithm 4.4 Bounded Well-Separated Pair Decomposition

```
1:  function COMPUTEWSPD($T, C_{\text{nodes}}, C_{\text{pairs}}$)
2:      for all $u \in$ innerNodes($T$) do
3:          if numPoints($u$) $> C_{\text{nodes}}$ then
4:              for all $(v, w) \in$ orderedPairsOfChildren($u$) do
5:                  COMPUTEWSPD($v, w, C_{\text{pairs}}$)
6:              end for
7:          else
8:              $D_{\text{nodes}} \leftarrow D_{\text{nodes}} \cup \{u\}$
9:          end if
10:     end for
11: end function
```

Chapter 5

The Fast Multipole Embedder

In this chapter, all the previous described pieces are put together and result in a force-directed layout algorithm named the *Fast Multipole Embedder (FME)*. First, the basic structure with the different steps will be explained, then we present how this can be done in parallel using multiple threads. The algorithm roughly consists of two phases:

First, a preprocessing step in the beginning, which calculates only the attractive forces and applies them by moving the nodes accordingly. In our implementation, this step is repeated 20 times regardless of any other conditions. The idea is to provide a better initial placement for the main step without using the expensive repulsive force calculation of the main step. However, in the following the preprocessing step is skipped, because the main step covers it.

Second, the main step follows the basic scheme of every force directed layout algorithm. In each iteration the repulsive and attractive forces are calculated, and then applied as a displacement vector. This is repeated until a fixed number of iterations is done or the maximum force acting on a single node falls under a given threshold. In our implementation we use the same repulsive force function like in [Hac05], because the multipole framework is based on it. Like mentioned in the introduction, the attractive force function differs and we use:

$$\vec{F}_{\mathrm{attr}}(u) = \frac{1}{\deg(u)} \sum_{(u,v)\in E} (p_u - p_v) \cdot \log \frac{|p_u - p_v|}{e_{uv}}$$

Dividing by the degree of the node relieves us from implementing a mechanism to prevent oscillation or other unpredictable behavior. Furthermore, the simulation seems to converge faster, thus fewer iterations are needed. In order to calculate the attractive forces, we do not iterate over the nodes. Instead, we calculate for each edge the force vectors for the two incident nodes and add these to the current force of the nodes. Obviously, the total running time of the attractive force computation is bounded by $O(|E|)$. In difference to the attractive force calculation, the repulsive force approximation is more difficult. Therefore, we describe it in a separate section.

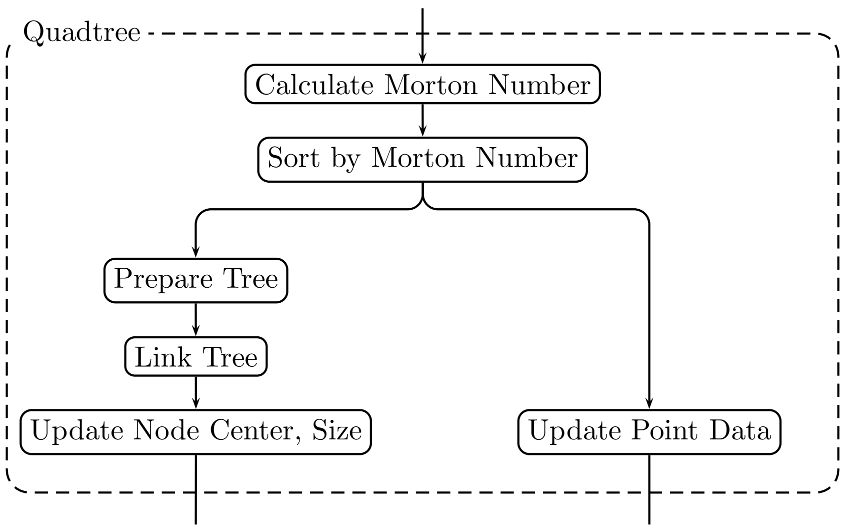


Figure 5.1: Detailed excerpt of the quadtree construction phase.

5.1 Repulsive Force Approximation

In order to approximate the repulsive forces in an iteration, we will follow the basic scheme:

1. Reduced quadtree construction (Chapter 3)
2. Well-separated pair decomposition (Chapter 4)
3. Repulsive force approximation using the Fast Multipole Method (Chapter 2)
4. Direct repulsive force evaluation

The first task is the construction of the reduced quadtree.

Like described in Chapter 3, first the morton number for each point is calculated, and then we sort the points by their morton numbers. This results in the fact that we have to deal with two different orders:

- The *graph order*, where the index i describes the i-th node in the graph.
- The *tree order*, defined by the morton order.

It will become clear later that the entire repulsive force computation can be done based on the tree order and does not require the graph order. Therefore it is useful to keep a copy of the point data, like coordinates and size, in this order. To avoid sorting the complete point data, an array is used which contains for each point only the morton number and the index of the node in the graph order. The copy of the point data is stored in a separate array and is updated after the morton numbers are sorted. We refer to this step as the *Update Point Data* step. The update is done by iterating over the sorted array and using the index which refers back to the original data.

This procedure has another side effect: In the next iteration we can reuse the array with the presorted morton numbers as the initial permutation for the sorting algorithm. Since this order "does not change much" after moving the nodes at the end of an iteration, we can benefit from this by using a sorting algorithm which is faster on a nearly sorted order.

Furthermore, the quadtree construction method only requires the sorted morton numbers as input, and not the coordinates of the points. As a result we can defer the update and build the quadtree first by preparing the tree nodes and adjusting the links. The center and size of each quadtree node are then calculated in the so called *Update Node Center, Size* step as described in Chapter 3. All these steps can be done in $O(n)$ time except for the sorting which requires $O(n \log n)$ time, where n is the number of points or graph nodes. Figure 5.1 displays an overview of the entire quadtree construction phase.

In order to use the Fast Multipole Method from Chapter 2, we first have to calculate the well-separated pair decomposition from Chapter 4. We use Algorithm 4.4 that takes time $O(n)$ and delivers, besides the decomposition W, the two sets D_{pairs} and D_{nodes}.

All the pairs and nodes in D_{pairs} and D_{nodes} have to be evaluated directly. For each node $v \in D_{\text{nodes}}$, we have to do a direct evaluation of the forces acting between the points contained in v. Since the number of points contained in v is bounded by C_{nodes} this takes time $O(C^2_{\text{nodes}})$. With the same argument we can bound the time needed for a pair of D_{pairs}. Furthermore, since Algorithm 4.4 terminates earlier than the original one, the number of pairs in D_{pairs} is $O(n)$. The size of D_{nodes} is bounded by $O(n)$ as well, because the quadtree has only $O(n)$ nodes. Thus, the total time needed for the direct evaluation step is $O(n \cdot (C^2_{\text{nodes}} + C^2_{\text{pairs}}))$. The resulting force vectors are stored in an array in tree order. We will map them back to graph order after the Fast Multipole phase is finished.

We can now use the theorems and lemmata from Chapter 2 together with the reduced quadtree and the well-separated pair decomposition to approximate the rest of the repulsive forces. Like in [Hac05, GR87] and most of the other works using the Fast Multipole Method, the following steps are done:

1. P2M (Point-to-Multipole) pass: Computes for each leaf the p-term multipole coefficients by using the multipole expansion theorem. The pass reads the point data (position and size), the leaf center, and modifies the multipole coefficients. This takes time $O(n \cdot p)$ for all points.

2. M2M (Multipole-to-Multipole) pass: Computes for each inner node the multipole coefficients by traversing the tree bottom up. At each node, the p-term multipole coefficients of the children are translated to the center of the current cell and then added to the existing ones. Each translation takes time $O(p^2)$, resulting in a total running time of $O(n \cdot p^2)$.

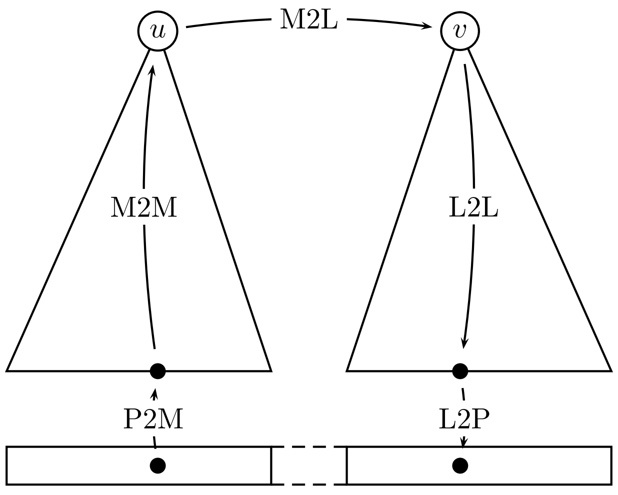


Figure 5.2: The five passes of the multipole framework and how a well-separated pair $(u, v) \in W$ is used to approximate forces acting on the points of v due to the points of u.

3. M2L (Multipole-to-Local) pass: The final approximation pass. For each pair $(u, v) \in W$ we convert the multipole coefficients of u to local coefficients of v and vice versa. The alternative way to do this is to iterate over all well-separated nodes of a node and convert their multipole coefficients, which has the advantage that only the local coefficients of this particular node are modified. From Chapter 4 it follows that the total size of the WSPD is $O(n)$ and since conversion takes time $O(p^2)$ the total time needed for this pass is $O(n \cdot p^2)$.

4. L2L (Local-to-Local) pass: Accumulates the local coefficients for the leaves by a top down traversal of the tree. For each node the shifted local coefficients of the parent are added. Like in the M2L pass, the time needed is $O(n \cdot p^2)$.

5. L2P (Local-to-Point) pass: For each point the forces induced by the fast multipole approximation are evaluated by using the local coefficients of the corresponding leaf. For all points, this takes time $O(n \cdot p)$.

Figure 5.3 shows a detailed outline of the different passes and their dependencies to the previous stages. Obviously, the running time of the multipole phase is $O(n)$ for a fixed p.

The result of both the multipole and direct evaluation are the repelling force vectors for each point. Since the vectors are needed when moving the graph nodes at the end of the iteration, they have to be stored in graph order rather than in tree order. Therefore, we map them back using the reference stored in the array we sorted in the beginning, which takes time $O(n)$ for all points.

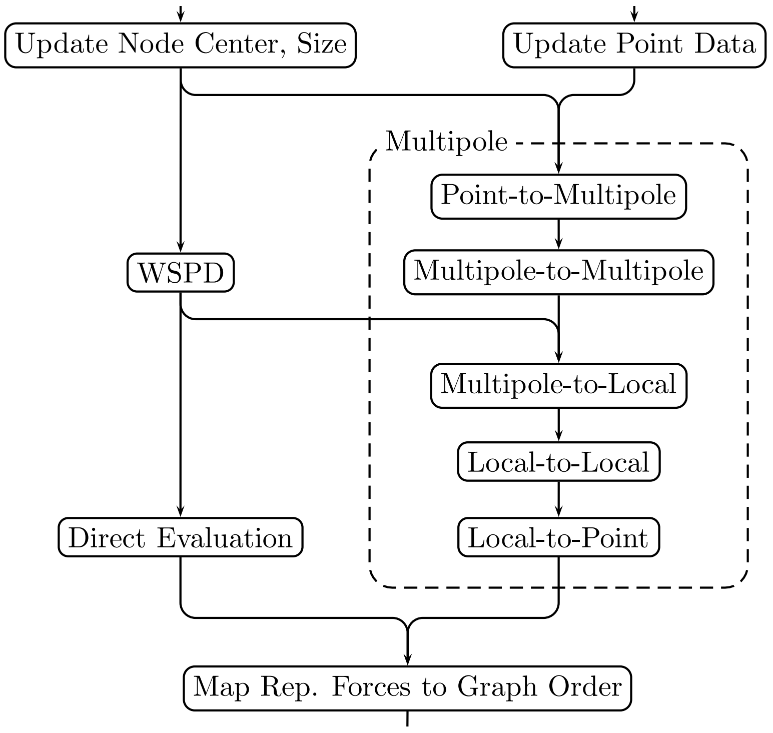


Figure 5.3: Detail excerpt of the Fast Multipole phase

In the last step of an iteration, the previously calculated forces are applied by moving the nodes. In our implementation we just multiply the force vector with the time step and use it as a displacement vector. Therefore this step can be done in time $O(n)$.

We summarize the running time of the repulsive force computation. For

- a fixed bitlength D during the quadtree construction,
- a fixed number of coefficients p in the multipole step,
- two constants C_{nodes} and C_{pairs} used by the WSPD,
- a point distribution such that the number of points in each leaf of the reduced quadtree is less than $\min\{C_{\text{nodes}}, C_{\text{pairs}}\}$,

the total running time is $O(n \log n)$. Furthermore, every step except the sorting by morton numbers takes time $O(n)$.

From the fact that n equals the number of nodes $|V|$ in the graph and that the calculation of the attractive forces due to the edges takes time $O(|E|)$ it follows that:

Corollary 5.1.1. *An iteration of the main step takes time* $O(|E| + |V| \cdot \log |V|)$.

Before we focus on the computation in parallel, we refer the reader to Figure 5.4 where all previously described steps and their dependencies are shown.

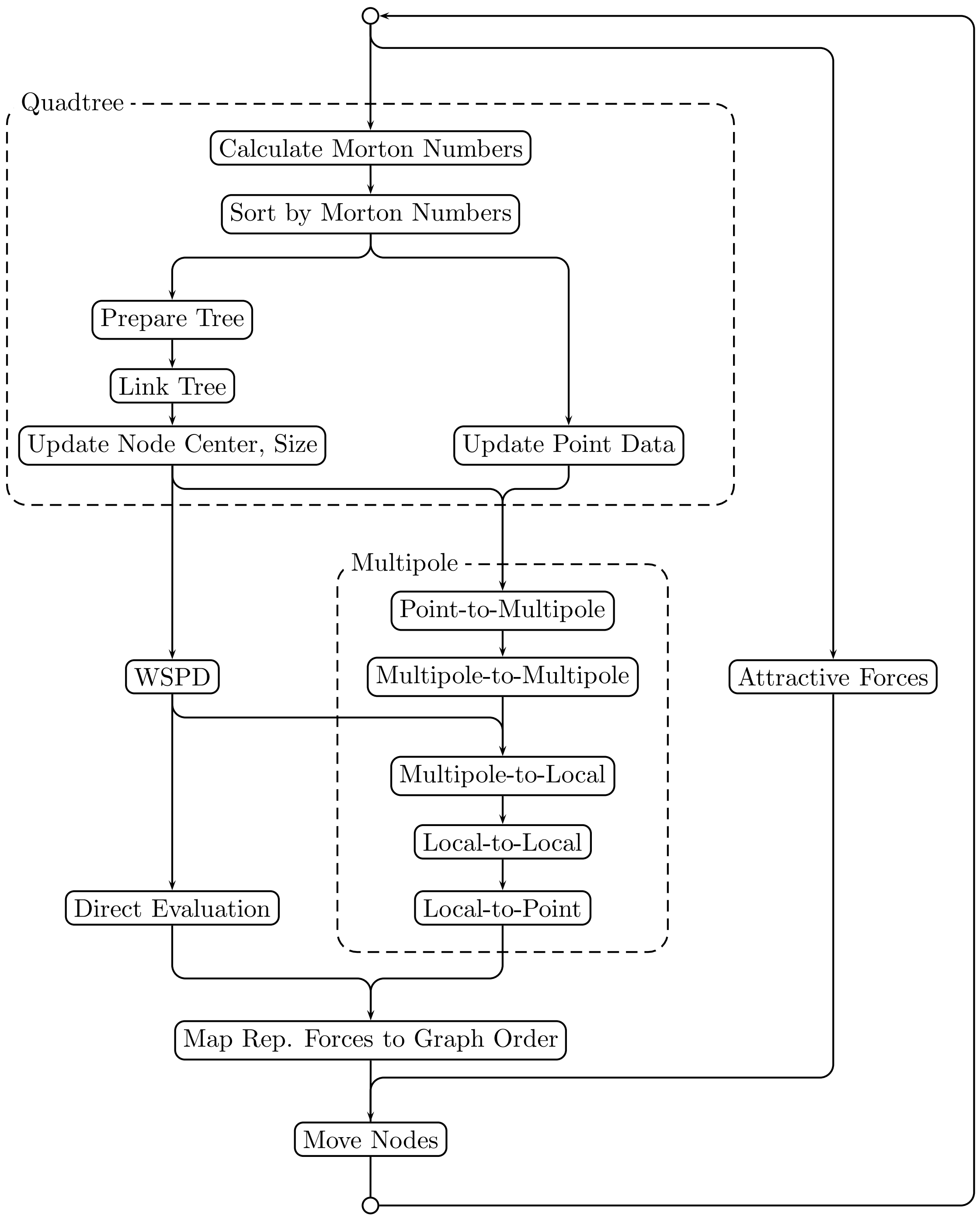


Figure 5.4: Detail excerpt of the steps in one iteration

5.2 Parallel Computation

In order to benefit from mulitcore architectures, the previously described algorithm has to be executed in parallel. Therefore, we first give a brief overview of problems arising during the procedure.

Many of the above described steps, e. g., the calculation of the morton number, are simple array or sequence operations. Such an operation can be characterized in the following way:

- Each operation depends only on the element it is called for and not on any other element in the sequence.
- The computational cost for each element is roughly the same.
- The result from one element is stored at a location in memory, which is exclusive for this element or the thread the element is assigned to.

Any operations that meet these requirements can be easily executed in parallel by partitioning the sequence into equally sized intervals, which are then assigned to the threads.

Most of the steps during the quadtree construction process can be classified as a simple array operation. The aforementioned morton number computation includes the computation of the bounding box, which is a simple min-max operation. Basically, each thread computes a bounding box for the points assigned to it, then the main thread uses these bounding boxes to obtain the bounding box for all points. The calculation of the grid coordinates and the interleaving of bits is then a point exclusive operation, because it uses as input the point coordinates and bounding box and only stores the resulting morton number for the point.

The preparation of the tree nodes is a little bit different, because we need to partition the array of points so that the boundaries of the interval do not divide a grid cell containing multiple points. In order to do this we use the standard partitioning and adjust the interval boundaries so that two points with the same morton number belong to the same partition. Then the tree nodes can be prepared in parallel. However, the linking of the tree nodes (Algorithm 3.4) is not executed in parallel, because it is very fast in practice and would require a lot of synchronizations due to the fact that it is a recursive function. The algorithm destroys the chain of inner nodes and leaves that we constructed earlier in the tree preparation step. Therefore, we restore it by a simple tree traversal in order to obtain a kind of linked-list of all nodes.

The obtained list is used to run the node update in parallel as simple sequence partition as well. On the other hand, the point data update is a simple array operation and does not require any special treatment.

When considering the calculation of the attractive forces, the problem arises that the resulting vectors for the source and target node have to be stored at a location in memory

Algorithm 5.1 Tree Partitioning

```
1: function PARTITIONTREE($v, L_{\text{par}}$)
2:     if numPoints($v$) $< \lfloor n/p^2 \rfloor$ then
3:         $L_{\text{par}} \leftarrow L_{\text{par}} \cup \{v\}$
4:     else
5:         for all $w \in$ children($v$) do
6:             PARTITIONTREE($v, L_{\text{par}}$)
7:         end for
8:     end if
9: end function
```

that may be accessed by multiple threads. The alternative is to use the nodes as a sequence and collect all forces due to incident edges. In that case the computational cost depends on the degree of the node which might differ a lot depending on the input graph. In order to keep things simple and to avoid difficult scheduling problems, we allocate for each thread a so called *local force array*, where we can store force vectors during an iteration. At the end of each force calculation, we add the forces calculated by the different threads and store them in the *global force array*, which is a simple sequence operation itself. Furthermore, the concept of the *local force array* allows us to use it in tree order during the repulsive force approximation. At the end of the repulsive force approximation, when we map the forces back to the graph order, we add all forces calculated by the threads and store the resulting vector at the corresponding position in the global force array in graph order.

Another important class of operations are the ones that are tree based, like a bottom-up or top-down traversal of the quadtree during the M2M pass or L2L pass. In order to run these in parallel we divide the tree into a serial and parallel part. The serial part consists of the top of the tree, whereas the parallel part consists of a number of distinct subtrees which are evenly distributed among the threads. The arising problem is that on the one hand the serial part should be as small as possible, but on the other hand, the distinct subtrees have to be small enough in order to distribute them fairly. This problem, however, is quite difficult and one can easily construct an example where such a partitioning becomes worthless, because the serial part consists of $O(n)$ nodes. Therefore, we will assume that the points are somehow evenly distributed in order to apply an easy and fast approach.

Given p threads, n points, and a reduced quadtree T, we recursively calculate a set of nodes containing less than $\lfloor n/p^2 \rfloor$ points by just splitting a node until we find nodes that contain a sufficient small amount of points. Algorithm 5.1 describes the procedure in detail. Each node visited by Algorithm 5.1 and not added to the set L_{par} belongs to the serial part that is rooted at the root of the tree. The nodes contained in L_{par} represent the roots of the distinct subtrees which have to be distributed among the threads. Every node in L_{par} contains less than $\lfloor n/p^2 \rfloor$, thus the number of nodes in L_{par} is at least p^2.

It remains the problem, how we distribute the previously calculated set among the threads such that the total number of points is roughly the same for each thread. There are several ways to do that, e. g., a first-fit or best-fit strategy can be used.

However, when considering the memory layout of the quadtree presented in Chapter 3, where all points contained in a subtree form a continuous interval in memory, it becomes clear that it is more beneficial to assign a thread the subtrees in the order in which they are discovered by Algorithm 5.1. Therefore, we assume that the list of children and L_{par} in Algorithm 5.1 is ordered and we distribute the nodes in L_{par} by the number of points contained in their subtrees using the following strategy: Each thread is assigned at most

$$\left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{1}{2} \cdot \frac{n}{p^2} \right\rfloor$$

points in total, except for the last thread which we assign the rest. As long as the current subtrees point count fits, we assign it to the current thread. Otherwise, we continue with the next thread.

Since the upper and lower bound for the number of points of the last thread are not that obvious, a more detailed description is given. First, for the lower bound we assign the maximum number of points to the first $p-1$ threads, then the rest left over for the last thread is

$$\begin{aligned} n-(p-1)\cdot\left(\left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{1}{2} \cdot \frac{n}{p^2} \right\rfloor\right) &\geq n-(p-1)\cdot\left(\frac{n}{p}+\frac{1}{2}\cdot\frac{n}{p^2}\right) \\ &= \frac{1}{2}\left(\frac{n}{p}+\frac{n}{p^2}\right). \end{aligned}$$

For the upper bound, we assign each thread the minimum number of points, which is

$$\begin{aligned} \left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{1}{2} \cdot \frac{n}{p^2} \right\rfloor - \left(\left\lfloor \frac{n}{p^2} \right\rfloor - 2\right) &\geq \frac{n}{p}-1+\frac{1}{2}\cdot\frac{n}{p^2}-1-\left(\frac{n}{p^2}-2\right) \\ &= \frac{n}{p}-\frac{1}{2}\cdot\frac{n}{p^2}. \end{aligned}$$

Otherwise, one more subtree would fit in the partition, because each subtree in L_{par} contains at most $\lfloor n/p^2 \rfloor - 1$ points. Thus, the rest left over for the last thread is bounded by

$$n-(p-1)\cdot\left(\frac{n}{p}-\frac{1}{2}\cdot\frac{n}{p^2}\right) \leq \frac{1}{2}\left(\frac{3n}{p}-\frac{n}{p^2}\right).$$

The runtime of this algorithm can only be bounded by $O(n)$, because one can construct an unbalanced binary tree which looks like a chain. In this case the algorithm has to add nearly all leaves along the chain.

As a result, the bottom-up traversal used in the M2M pass can now be run in parallel. Since the P2M pass modifies the p-term multipole coefficients of the leaves, we execute this step when the bottom-up traversal reaches a leaf.

The M2M pass is done by a top-down traversal that follows the basic principle. In difference to the P2M pass, the L2P pass does not modify any tree node related data. Therefore, we can classify it as a simple point array operation.

It remains the well-separated pair decomposition in combination with the M2L pass and the direct evaluation step. Algorithm 4.4 iterates over all inner nodes and computes a decomposition for the corresponding subtree. Doing this in parallel with only a few synchronizations is quite difficult. However, in the following we present a simple and fast solution. Instead of iterating over all nodes we divide the procedure into two parts.

First, the serial part where the main thread iterates over all nodes contained in the serial part of the tree partitioning and starts the recursive Algorithm 4.3 for all ordered pairs of children. The result is a partial decomposition W' and two partial sets D'_{pairs} and D'_{nodes} for the serial part of the tree. We run the M2L pass on W' to obtain the local coefficients and do a direct force calculation for D'_{pairs} and D'_{nodes}.

In the second part, the above described procedure is repeated by the threads for the subtrees rooted at L_{par}. All recursive calls made during the well-separated pair decomposition (Algorithm 4.3) affect only nodes which are contained in the subtree rooted at the node from which the call was initiated. This ensures that both, the affected node and point sets are distinct because the subtrees rooted L_{par} are distinct.

The above described solution leaves room for improvements, because the work which is done in the serial part can be quite a lot. However, experiments have shown that this approach is superior to the alternative solution, which is to compute the WSPD with a single thread and to run only the M2L pass and the direct evaluation step in parallel.

Chapter 6

The Fast Multipole Multilevel Embedder

Like shortly mentioned in the introduction, a so called *multilevel method* can be used to determine a good initial placement of the nodes. The basic idea is to create multiple graphs $G_1, \ldots, G_m$ from the original graph $G = G_0$ with different resolutions. In order to achieve this, a coarser graph G_{t+1} is created based on the finer graph G_t, which serves as input graph for the next level. This is repeated until the actual graph G_m contains only a fixed number of nodes or some other condition is met.

This phase is called the *coarsenning phase*, in which a partitioning strategy is deployed to create the coarser graphs. Common requirements such a strategy should met are:

1. The number of multilevels is bounded by $O(\log |V|)$.

2. The number of multilevels is not too small in order to find a good initial placement for the nodes.

3. The runtime of the coarsening phase is bounded by $O(|V_t| \cdot \log |V_t| + |E_t|)$

A force-directed layout algorithm is then used to calculate a layout for each level, starting with the coarsest level. This phase is referred to as *refinement phase.* At each step the previously calculated layout of the coarser graph G_{t+1} is used to find a good initial placement for G_t. This precoedure is repeated until a layout for G_0, the original input graph, has been calculated. Figure 6.1 displays both phases.

In the following, we first describe the coarsening phase and the strategy. The idea follows the basic principle of [Hac05]. Second, we give a brief description of the refinement phase. The resulting algorithm is called the *Fast Multipole Multilevel Embedder (FMME)*, which deploys the Fast Multipole Embedder, described in the previous chapter.

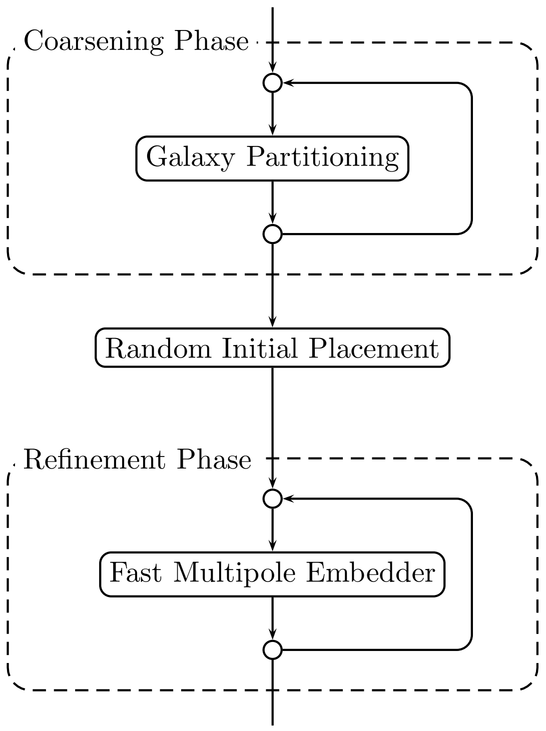


Figure 6.1: Excerpt of the Fast Multipole Multilevel Embedder

6.1 Coarsening Phase

As a partition strategy, a slightly modified version of the *galaxy partitioning* from [Hac05] is used. The name is based on the principle of selecting a set of nodes as *suns* and labeling adjacent nodes as *planets*. Nodes that are only adjacent to planets and not to a sun, are referred to as *moons*. Figure 6.2 shows a small example of a galaxy partitioning. Furthermore, two suns are not allowed to be adjacent and a planet must not be adjacent to more than one sun. A moon is adjacent to at least one planet. As a result each node is part of a so called *solar system*. An edge which connects two solar systems is referred to as *inter-system edge*, whereas the rest is called *inner-system edges*.

The basic idea is to label a graph G_t and construct a coarser graph G_{t+1} in which each solar system of G_t is represented by a node. If two systems are connected by at least one inter-system edge in G_t, then there exists an edge between the two nodes in G_{t+1} which represent the two systems. During the coarsening process each node has a property called the *mass* of the node. All nodes of the original graph G_0 have a mass of one, whereas the node mass in a coarser graph is defined as the sum of all masses contained in the system that the node represents.

The original approach described in [Hac05] creates the coarse graph by collapsing each solar system. Since we need the original graph that approach requires to make a copy of

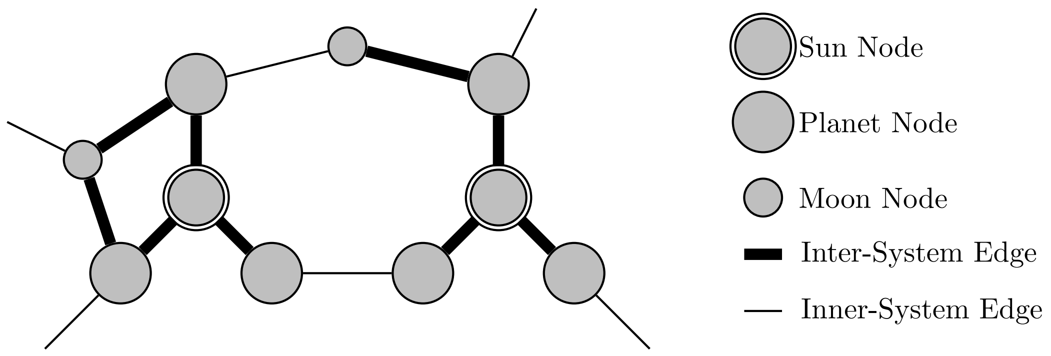


Figure 6.2: Example for a galaxy partitioning consisting of two systems

the input graph before we modify it. A sun is selected by a randomized algorithm that first chooses a set of nodes randomly and then selects the node with the lowest mass from the set. The principle of choosing a node with a low sun mass is important in order to prevent an unbalanced coarsening of the graph. The resulting algorithm is a linear time algorithm for partitioning.

In the following, an algorithm is presented without the need of a copy, instead it labels the nodes of the input graph and creates the output graph from scratch. Furthermore, we sacrifice the linear run time of the original algorithm in order to obtain a better sun selection. The algorithm is displayed in Algorithm 6.1 and works as follows:

At the beginning of each coarsening step, we estimate for each node the system mass if we would choose it as a sun. The estimation is done by adding the node's current mass and the mass of all adjacent nodes, since these would be the planets of the system, whereas possible moons are not taken into account.

$$m_{\text{est}}(v) = m(v) + \sum_{(v,w)\in E_t} m(w)$$

Then we sort all nodes by their estimated sun mass m_{est}. One problem arising during this step is caused by the initial permutation of the nodes. For example consider a regular grid graph, in the first coarsening step the estimated sun mass m_{est} of all nodes that are not part of the border is 5. When using a stable sorting algorithm these nodes occur in the same order like in the input graph. There might be some scenarios where this is an advantage, however, experiments have shown that it is usually not beneficial because the partitioning relies too much on the input permutation. For this reason we sort the nodes in the following way: The array that has to be sorted, consisting of the keys m_{est} and the references to the nodes, is initialized by randomly choosing the nodes from the original sequence.

Algorithm 6.1 Coarsening Step

```
1: function createCoarseGraph($G_t = (V_t, E_t), m : V_t \mapsto \mathbb{N}$)
2:     for all $v \in V_t$ do
3:         label($v$) $\leftarrow 0$
4:         $m_{\text{est}}(v) \leftarrow m(v) + \sum_{(v,w) \in E_t} m(w)$
5:     end for
6:     $a_1, \dots, a_n \leftarrow$ sort nodes of $G_t$ by $m_{\text{est}}$ in ascending order
7:     for $v \leftarrow a_1, \dots, a_n$ do
8:         if label($v$) $= 0$ then
9:             label($v$) $\leftarrow 3$
10:            sun($v$) $\leftarrow v$
11:            labelSystem($v, v, 2$)
12:        end if
13:    end for
14:    for all suns in $V$ do
15:        create a corresponding node in $G_{t+1}$ and calculate the exact node mass
16:    end for
17:    for all edges $e = (u, v) \in E_t$ with $sun(u) \neq sun(v)$ do
18:        create an edge in $G_{t+1}$ between the nodes representing $sun(u)$ and $sun(w)$
19:    end for
20:    remove parallel edges in $G_{t+1}$
21:    return $G_{t+1}$
22: end function
```

After sorting the nodes by their estimated sun mass, we iterate over the sorted order and assign the nodes their role. Each node is labeled with a number label(v) $\in \{0, 1, 2, 3\}$ which corresponds to $\{\textit{Not Labeled} = 0, \textit{Moon} = 1, \textit{Planet} = 2, \textit{Sun} = 3\}$. If the current node has not been labeled before, meaning the node is not a sun, planet, or moon, we choose it as a sun and start a recursive labeling procedure in order to label the rest of the solar system. Algorithm 6.2 describes the recursive labeling in detail and works as follows: The function labelSystem is first called for the sun itself. It iterates over all adjacent nodes and assigns them the label 2 (*Planet*). Note that some of these nodes might be moons of another system. In this case the moon is promoted to a planet and the solar system it belongs to is changed. However, none of these nodes can be a planet because the call labeling these as planets would have labeled the current sun as a moon, which is a contradiction to our assumption that the current sun has not been labeled before. Therefore, all nodes adjacent to our current sun have to be a moon or are not labeled yet. After assigning the node the planet label, a recursive call is made in order to label surrounding nodes as moons (label(v) $= 1$). This is only done for nodes which have never

Algorithm 6.2 DFS-Based Labeling of a Solar System

```
function labelSystem($s, v, l$)
    for all $e = (v, w) \in E$ do
        if label($w$) $< l$ then // if the node can be promoted
            if sun($w$) $\neq s$ then // and belongs to another system
                sun($w$) $\leftarrow s$ // then it becomes part of the current system
            end if
            label($w$) $\leftarrow l$ // promote it by assigning a greater label
            if $l > 1$ then
                labelSystem($s, w, l - 1$) // continue labeling
            end if
        end if
    end for
end function
```

been labeled before. An example for the labeling procedure is given in Figure 6.3. Since it is not obvious that the time needed by this procedure can be bound by $O(|V_t| + |E_t|)$, we formalize this in a lemma.

Lemma 6.1.1. *The labeling procedure of Algorithm 6.1 takes time $O(|V_t| + |E_t|)$.*

Proof. Before a recursive call inside labelSystem (Algorithm 6.2) is made, the source node has been promoted from either zero to a moon or from a moon to a planet. From the fact that the two incident nodes of an edge can only be promoted constant times, it follows that the edge is only used constant times by labelSystem and the bound for the time follows. □

When all nodes are labeled, we have to construct the coarser graph G_{t+1}. For each node that is labeled as a sun, we create a corresponding node in the coarser graph and calculate the exact mass. Then we add for each inter-system edge a corresponding edge to E_{t+1}, which may result in some parallel edges. As an example, consider Figure 6.3, where we obtain four inter-system edges incident to the two systems. Therefore, we remove all parallel edges from G_{t+1} at the end of Algorithm 6.1.

We can summarize the time needed for the coarsening step. At each level the sorting step takes time $O(|V_t| \log |V_t|)$, all other steps can be done in time $O(|V_t| + |E_t|)$, including the parallel edge removal. Furthermore, since we do not introduce any new edges, $|E_{t+1}| \leq |E_t|$ holds. Like in [Hac05] the number of suns is at most $|V_t|/2$ in each step, because each solar system consists of at least one sun and one planet. Thus, the total number of multilevels is bounded by $O(\log |V|)$ and each level requires time $O(|V_t| \log |V_t| + |E_t|)$,

Lemma 6.1.2. *The time needed by the coarsening step is $O(\log |V| \cdot (|V| + |E|))$.*

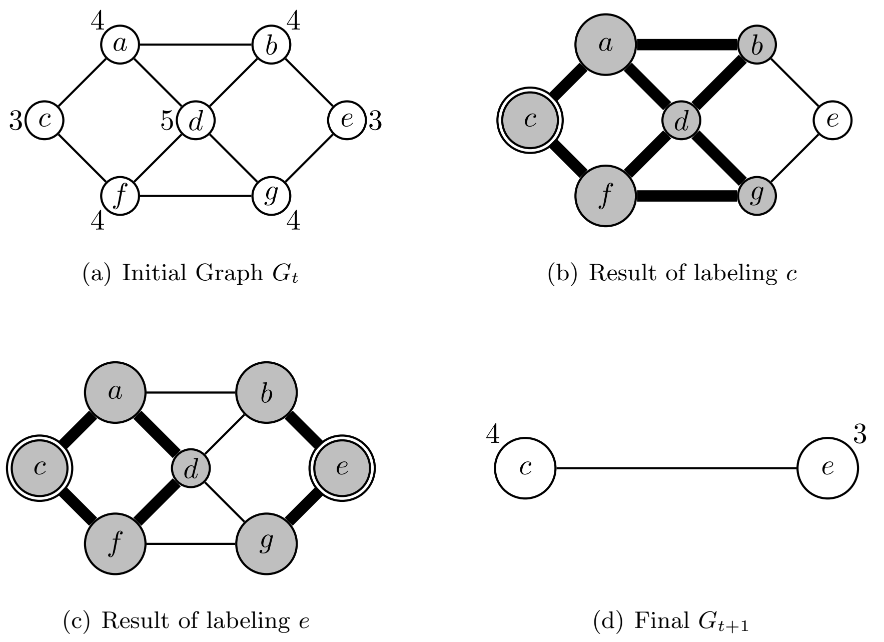


Figure 6.3: Small example for labelling a system. At the beginning a new system has been labeled with c as sun. In the second figure(b) the last remaining node is labeled as a sun. Clearly visible the two moons b, g become planets of e, whereas d remains a moon of c

Proof. From the fact that $|V_{t+1}| \leq |V_t|/2$ and that sorting takes time $O(|V_t| \log |V_t|)$ it follows that the time needed to sort the nodes of all $m + 1$ multilevels is bounded by

$$\begin{aligned}
\sum_{k=0}^{m} \frac{|V|}{2^k} \log \frac{|V|}{2^k} &= |V| \cdot \sum_{k=0}^{m} \frac{\log |V| - k}{2^k} \\
&= |V| \cdot \left(\log |V| \cdot \sum_{k=0}^{m} \frac{1}{2^k} - \sum_{k=0}^{m} \frac{k}{2^k} \right) \\
&< |V| \cdot 2 \log |V| \\
&= O(|V| \log |V|)
\end{aligned}$$

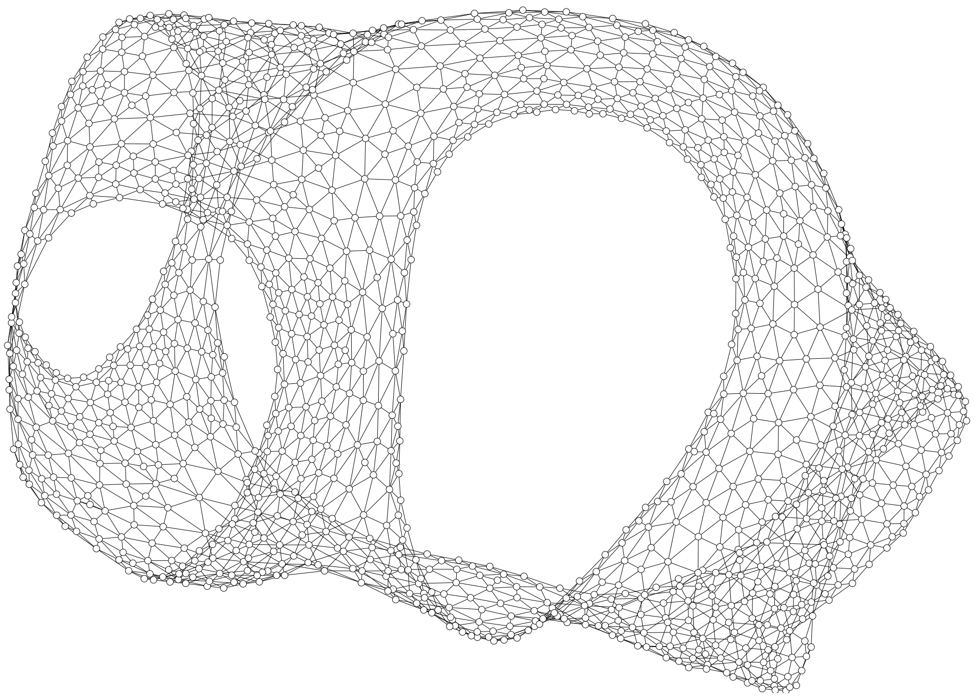
Thus, the total runtime is $O(\log |V| \cdot (|V| + |E|))$. □

6.2 Refinement Phase

In the refinement phase the actual layout of the graph is computed. At the beginning a random initial placement for the coarsest graph is generated and the Fast Multipole Embedder is used to layout that level. The resulting layout is then used to calculate a good inital placement for the next finer level before running the layout algorithm.

Unlike to the method used in [Hac05], we just place the nodes randomly near the position of their sun's representative in the coarser level. For one multilevel this can be done in time $O(|V_t|)$, resulting in a total time of $O(|V|)$. In addition the Fast Multipole Embedder consists of a preprocessing step, which calculates only the edge forces and moves the nodes accordingly. This is repeated for a fixed number of iterations, in order to find a good placement for the nodes before running the main step that includes the expensive repulsive force approximation. The total time needed by the Fast Multipole Embedder for all levels is bounded by $O(\log |V| \cdot (|V| + |E|))$. We do not need to prove the runtime, because the proof of Lemma 6.1.2 from the previous section deals with the same runtime for the coarsening step.

Furthermore, the layout is rescaled by a constant factor (in our implementation 1.4) to relieve the Fast Multipole Embedder of this task to scale during the simulation since it would cost a lot of iterations. Figure 6.4 shows an example of two resulting layouts during the refinement phase.


(a) Layout of G_1

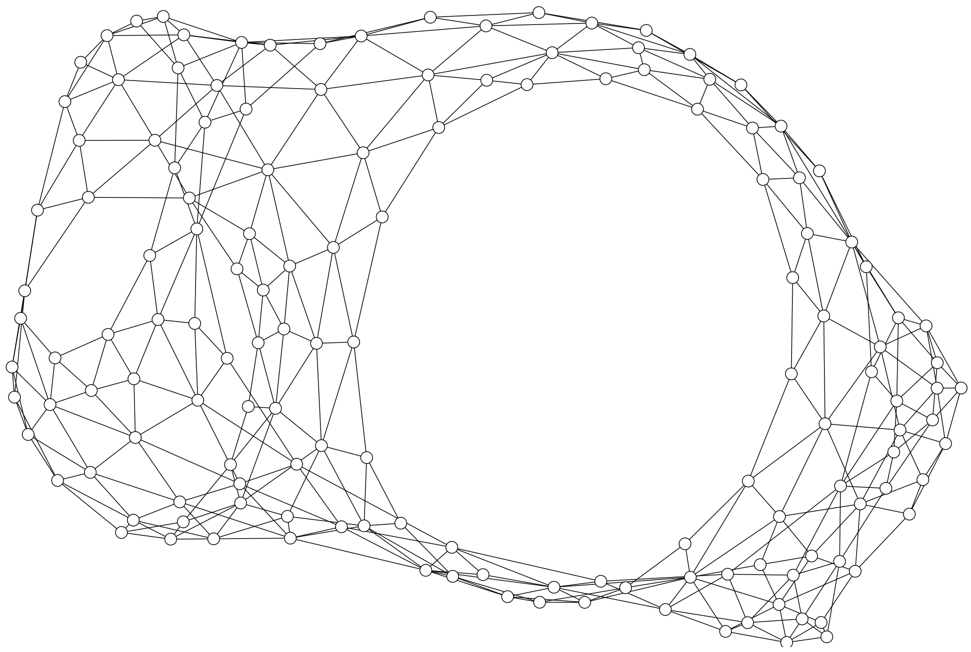

(b) Layout of G_2

Figure 6.4: Example layout of two levels, G_1 and G_2, during the refinement phase. The graph is fe_elt2 from the benchmark set.

Chapter 7

Experimental Results

In this chapter the results of the Fast Multipole Multilevel Embedder are presented. The first section deals with the test environment and gives some interesting insights into modern multicore systems. Then the actual layout algorithm is tested on a given set of commonly used benchmark graphs. Furthermore, a graph generator is used to investigate the runtime of the algorithm even for very large graphs with nearly one million nodes.

7.1 Test Environment

For all experiments a machine with two Intel Xeon E5430 processors clocked at 2.66GHz has been used. Each physical processor is a quadcore CPU (see Figure 7.1) and machine is running Ubuntu Linux with Kernel Version 2.6.24-16-generic. The compiler used is the GCC Version 4.2.3 (Ubuntu 4.2.3-2ubuntu7) with optimization level 1 (-O1) for the OGDF and level 3 (-O3) for the benchmark in this section. Before we present the results of the actual algorithm, the memory performance of the platform is investigated. Consider

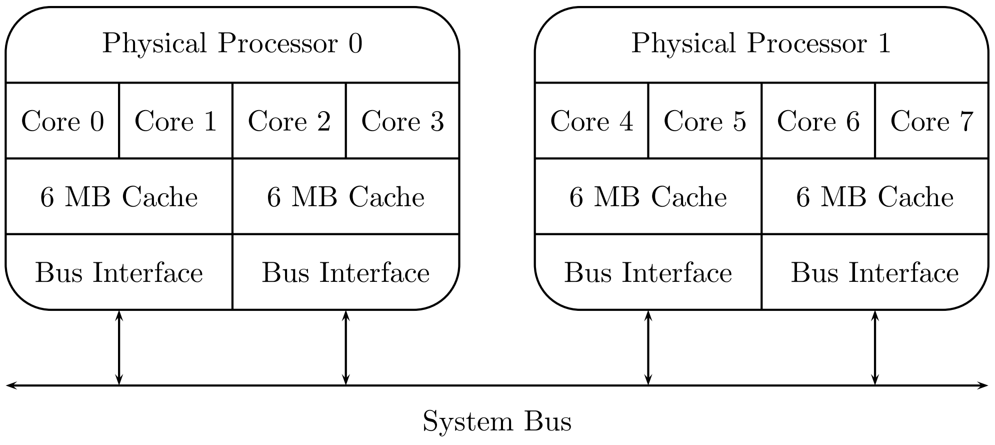


Figure 7.1: Dual Socket Intel Xeon E5430 schematic view.

the function which moves the nodes at the end of each iteration. Basically this function iterates over all nodes and modifies the position using only a few arithmetic instructions. The function runs in linear time and intuitively one would assume that the execution in parallel will result in a good speedup. But instead the speedup is limited by the memory bandwidth between RAM and processor. The reason for this is that the time needed for a small number of instructions on a modern CPU can be neglected compared to the time required to fetch the data from RAM. This function, like many others, just needs each array entry once, therefore the cache is useless and the bus to the memory becomes the bottleneck. Note that this is independent of the amount of data. In our application a graph might be so small that all data structures used during an iteration fit in the cache. However, in this case the overhead for running it in parallel becomes too high, and thus this case does not matter.

A simple way to measure the actual memory bandwidth between RAM and CPU is to write a small benchmark which creates a large array of floats (here 1 GB). Then we iterate over the array and add to each floating point value a constant. Therefore each element has to be loaded into the cache and stored back at the end. The resulting traffic is then two times the size of the array. Furthermore, this benchmark is repeated using multiple threads placed on different cores and on different physical processors for measuring the memory bandwidth. The test consists of using 1, 2, 4 and 8 threads which are then assigned to the cores using two different strategies. Considering Figure 7.1, the first strategy consists of assigning thread 0 to core 0, thread 1 to core 1, etc.

The second strategy scatters the threads as much as possible by placing them according to the following pattern:

	CPU 0				CPU 1			
Number of Threads	core 0	core 1	core 2	core 3	core 4	core 5	core 6	core 7
1	0							
2	0				1			
4	0		1		2		3	
8	0	1	2	3	4	5	6	7

Table 7.1: The pattern used in the scatter strategy

In addition the test is run with and without utilizing SSE. Instead of using OGDF's thread API, OpenMP is used and a thread is placed on the corresponding core using the `sched_setaffinity` function for setting the threads processor affinity. Note that the actual system core enumeration on which the affinity mask is based, differs from our enumeration. Table 7.2 displays the result of the benchmark, which are obtained by running the test for each configuration a hundred times and taking the best result. Obviously the use of SSE does not have a big impact on the performance except the single-

Number of Threads	In Order		Scattered	
	normal	with SSE	normal	with SSE
1	0.484	0.459	0.484	0.459
2	0.425	0.419	0.259	0.258
4	0.388	0.387	0.262	0.261
8	0.271	0.271	0.267	0.266

Table 7.2: Benchmark result times for adding a constant to each entry (1 floating point operation) of a 1 GB floating point array (≈ 268 million floating points).

threaded run where it performs slightly better. In the following the results without SSE acceleration are discarded, and a closer look at the two different strategies is taken instead. Since this chapter is not about measuring the performance of multicore architectures in general, but the performance of the Fast Multipole Multilevel Embedder, these results should only clarify the problems arising during the execution of the algorithm's code. However, the results are conclusive and can be used to explain the memory bandwidth bottleneck.

Number of physical CPUs	Number of Threads	Time in seconds	Speedup	Speed in Gflop/s	Bandwidth in GB/s	
					per core	total
1	1	0.459	1.00	0.58	4.36	4.36
	2	0.419	1.10	0.64	2.39	4.77
	4	0.387	1.19	0.69	1.29	5.17
2	8	0.271	1.69	0.99	0.92	7.37

Table 7.3: Bandwidth benchmark results using SSE and the first strategy where the threads are assigned to the cores in order.

Number of physical CPUs	Number of Threads	Time in seconds	Speedup	Speed in Gflop/s	Bandwidth in GB/s	
					per core	total
1	1	0.459	1.00	0.58	4.35	4.35
2	2	0.258	1.78	1.04	3.88	7.75
	4	0.261	1.76	1.03	1.91	7.65
	8	0.266	1.72	1.01	0.94	7.51

Table 7.4: Bandwidth benchmark results using SSE for the scatter strategy.

Table 7.3 displays the bandwidth and the resulting CPU speed when using the first strategy. Only when using eight threads, the second processor is used. Interestingly, even when using four threads on the first processor, the system is still not able to achieve the maximum bandwidth. The technical specification of the processor state a bus bandwidth of 10.6 GB/s, which is not nearly achieved in any of the tested configuration. The practical maximum bandwidth is about 7.7 GB/s in this benchmark. However, it seems the two processors have to share the bus bandwidth at a fixed rate and therefore each processor can only achieve a maximum bandwidth of about 5.3 GB/s. The results of the scattering strategy shown in Table 7.4 support this claim. The best throughput of all configurations is achieved using two threads running on different physical processors and, therefore, are not limited in bandwidth. This value is the maximum and governs the performance, when increasing the number of used cores. As a conclusion it can be said that the overall performance of this rather simple task on this particular machine is not good. When considering the achieved speedup factor it becomes clear that this type of platform performs good for tasks which use less memory but require a lot of computational power. Furthermore, when considering Table 7.2 it becomes clear that using SSE does not solve this problem.

7.2 Runtime

In the following the results of the experiments with the Fast Multipole Multilevel Embedder are presented and compared to the results of the FM^3 and the graphics processor (GPU) based algorithm from [GHGH09]. Both, our FMME and the FM^3 implemention are part of OGDF and compiled with the same options to guarantee a fair test. The resulting times for the GPU implementation have been taken from the corresponding article[GHGH09], because the implementation requires special hardware.

The graphs are taken from [Wal, Hac05] and are commonly used as benchmark instances for layout algorithm. The selection of graphs from the Walshaw Graph Collection [Wal] is based on the size of the instances, in order to test our algorithm for large inputs. Furthermore, we selected some artificial generated graphs from [Hac05] to test the scalability of our algorithm.

Unlike FM^3, our algorithm requires that the graph is connected. Since two of the benchmark graphs (fe_body and bcsstk29) contain more than one connected component, a module from OGDF is used that calls for each connected component the FMME and packs the resulting layouts. However, we have not selected any graphs with a large number of connected components, because each component has to be run separately and our algorithm is not optimized for small instances.

The number of coefficients during the multipole step is 4 for the FMME and the GPU algorithm, whereas the FM^3 is set to run as fast as possible, which corresponds to a count of 2. Furthermore, if not stated otherwise, the maximum number of threads for the FMME is set to eight and SSE is used during the multipole phase. The threads are assigned to the cores by using the scatter strategy from the previous section.

Table 7.5 shows the results for the FMME, the FM^3, and if available, for the GPU-based algorithm. The results show that our algorithm is superior in runtime for all instances.

Graph Information				Runtime		
Type	Name	$\|V\|$	$\|E\|$	FMME	FM^3	GPU
Artificial Generated	grid_rnd_010	97	169	0.015	0.039	
	grid_rnd_032	985	1834	0.120	0.460	
	grid_rnd_100	9497	17849	0.328	5.392	1.72
	grid_rnd_320	97359	184532	1.187	68.197	
	cylinder_rnd_010_010	97	178	0.016	0.040	
	cylinder_rnd_032_032	985	1866	0.158	0.453	
	cylinder_rnd_100_100	9497	17941	0.321	5.595	
	cylinder_rnd_320_320	97359	184821	1.202	67.523	
	sierpinski_04	123	243	0.044	0.063	
	sierpinski_06	1095	2187	0.215	0.486	
	sierpinski_08	9843	19683	0.285	4.535	0.984
	sierpinski_10	88575	177147	1.014	49.925	
	flower_005	930	13521	0.265	0.383	
	flower_050	9030	131241	0.279	3.874	0.547
	flower_500	90030	1308441	1.561	47.684	
	spider_A	100	160	0.039	0.048	
	spider_B	1000	1600	0.398	0.531	
	spider_C	10000	22000	0.328	4.625	1.49
	spider_D	100000	220000	1.340	54.255	
Walshaw Graph Collection	bcsstk29	13992	302748	0.947	11.094	
	bcsstk31_connected	35586	572913	0.977	27.612	1.31
	bcsstk32	44609	985046	1.382	37.722	1.99
	bcsstk33	8738	291583	0.640	8.003	0.968
	4elt	15606	45878	0.362	9.377	1.58
	crack	10240	30380	0.365	5.635	0.937
	data	2851	15093	0.234	1.330	
	t60k	60005	89440	0.836	41.398	
	fe_body	44775	163734	4.165	29.806	
	fe_4elt2	11143	32818	0.336	6.482	
	fe_pwt	36463	144794	0.589	21.035	2.48
	fe_ocean	143437	409593	2.092	105.989	7.97
	dg_1087	7602	7601	1.513	4.546	

Table 7.5: Selected Graphs and the runtime needed to calculate the layout.

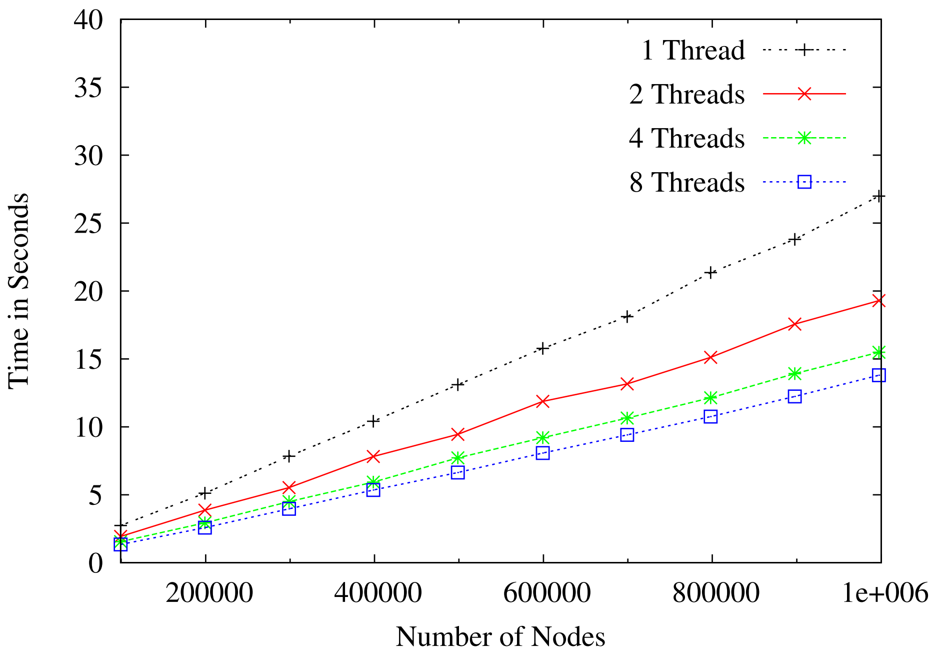


Figure 7.2: The runtime of the FMME for a series of grid graphs when using multiple threads.

When considering the small graphs it becomes clear that our algorithm does not perform so well in comparison to the bigger instances. Almost every graph of the artificial generated ones (flower, spider, . . .) with approximately 1000 nodes takes the same time as one from the same type but ten times the size.

For bigger instances, e. g. fe_ocean, the algorithm benefits from the parallel execution and the array-based quadtree construction. The factor for fe_ocean compared with the FM^3 and GPU implementation increases up to 50 and 3.8, respectively.

In addition to the above described graphs, we implemented a random grid graph generator in order to be able to test graphs larger than the ones in the benchmark set. The grid graphs are generated by first creating a regular square grid, then 3% of the nodes are randomly deleted. In case the graph is no longer connected, additional edges are inserted to obtain a connected graph. The generator is used to measure the runtime for very large graphs depending on the number of cores used. Figure 7.2 shows the time for a series of grids ranging from approximately one hundred thousand to one million nodes. The corresponding speedup factor is shown in Figure 7.3. The maximum achieved speedup factor is close to 2.0 when using all eight cores.

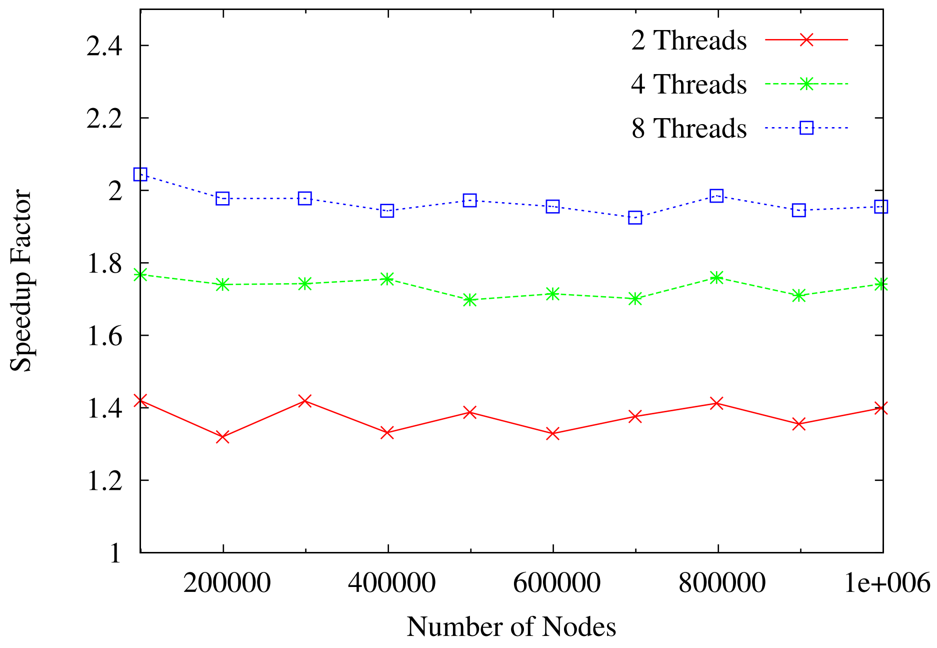


Figure 7.3: The resulting speed up for the grid graphs.

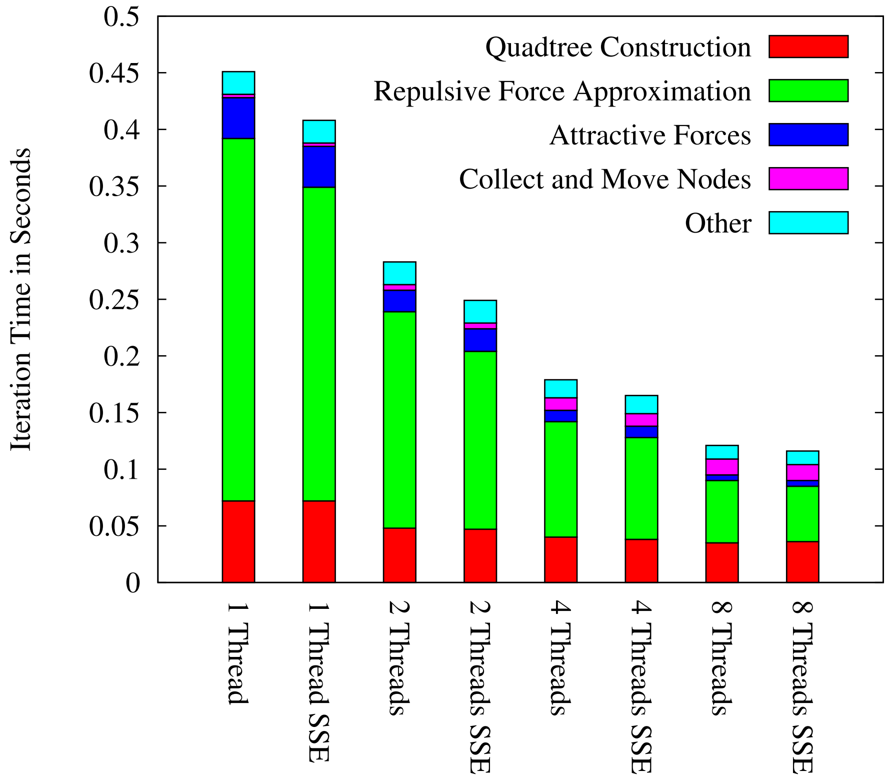


Figure 7.4: The time spent in the different phases during an iteration of the FME for fe_ocean with 143437 nodes and 409593 edges.

In order to give an impression how the runtime is distributed during an iteration of the Fast Multipole Embedder, we pick fe_ocean as an example graph and analyze the time needed by the different phases during one iteration. Figure 7.4 shows the time spent in the different phases depending on the number of threads used.

The times are captured during a random iteration, while calculating the layout of the finest multilevel with a size of $|V| = 143437$ and $|E| = 409593$. Furthermore, the time is measured for a run with and without using SSE for the multipole steps.

The quadtree construction includes everything from the calculation of the morton numbers to the Point Update and Node Center, Size step. The repulsive force approximation step consists of the well-separated pair decomposition, the direct force evaluation, the five passes of the fast multipole method, and the step which collects the repulsive forces and maps them back to the graph order. The time needed to calculate the edge forces is represented by the attractive force step. In the last step, the previously calculated attractive forces are collected and added to the repulsive forces, then the nodes are moved by the displacement vector. Anything else, like reseting variables, capturing these results, and so on is represented by the category named other.

Clearly visible is the price we have to pay for the local force array concept. The runtime for the collect and move step increases with the number of used cores. However, this concept simplifies the edge force calculation, which results in good speedup for the attractive forces.

The benefits of SSE are noticeable and do not stagnate while the number of threads increases. Together with an acceptable speedup factor of the approximation phase, this indicates that the required amount of time is governed by the calculations and not by the memory bandwidth.

Unlike the approximation phase, the quadtree construction consists of many computationally cheap, but memory expensive steps, which results in a speedup factor below two.

As a conclusion it can be said that, when considering the memory bandwidth bottleneck of the test system, the scalability of the Fast Multipole Embedder is acceptable, especially for large input sizes. However, from the fact that the algorithm is called for small instances during the refinement phase of the multilevel step, it follows that the speedup decreases when it comes to the total runtime of FMME.

7.3 Resulting Layouts

In the following the drawings calculated by our algorithm are displayed. All drawings have been rotated manually to fit better into the document. Furthermore, the edges in graphs with a high edge-node ratio, e. g. bcsstk33, are drawn with a lesser opacity in order to display the structure more clearly.

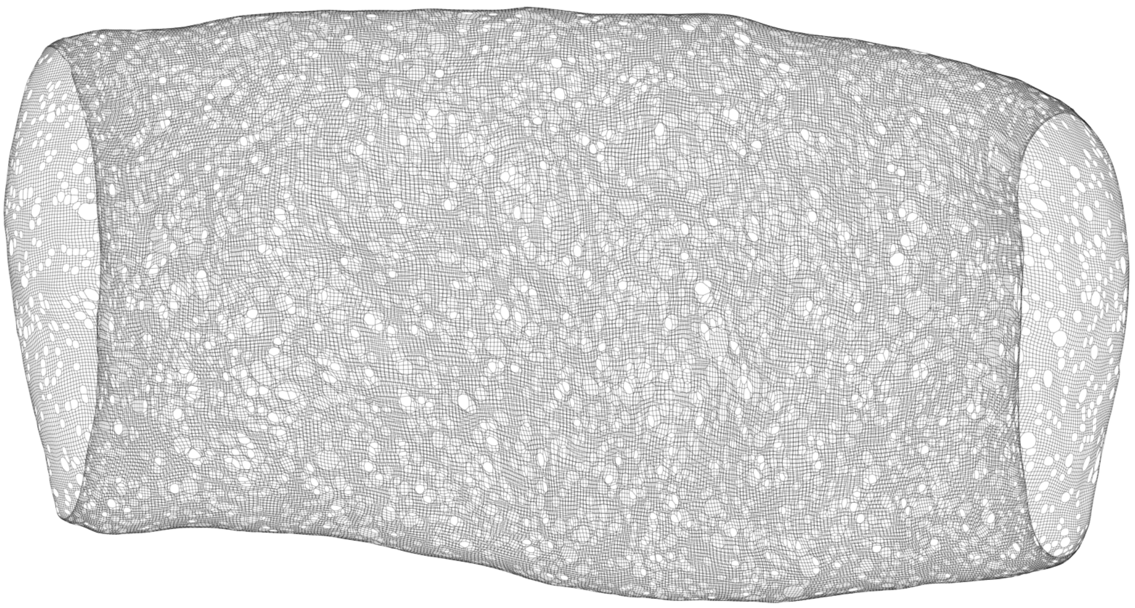


Figure 7.5: Layout of cylinder_320_320 with 97359 nodes and 184821 edges (1.202s).

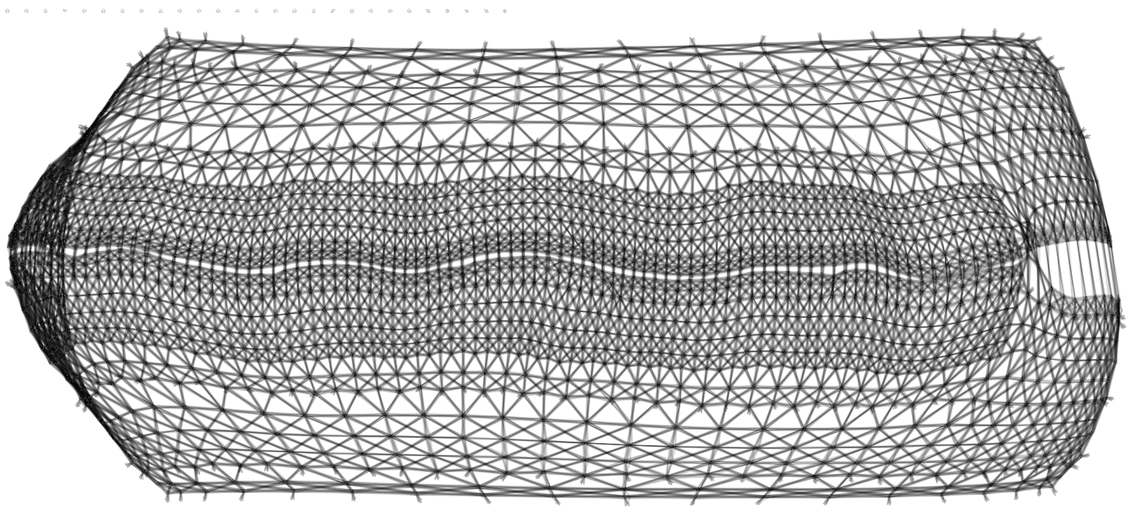


Figure 7.6: Layout of bcsstk29 with 13992 nodes and 302748 edges (0.947s).

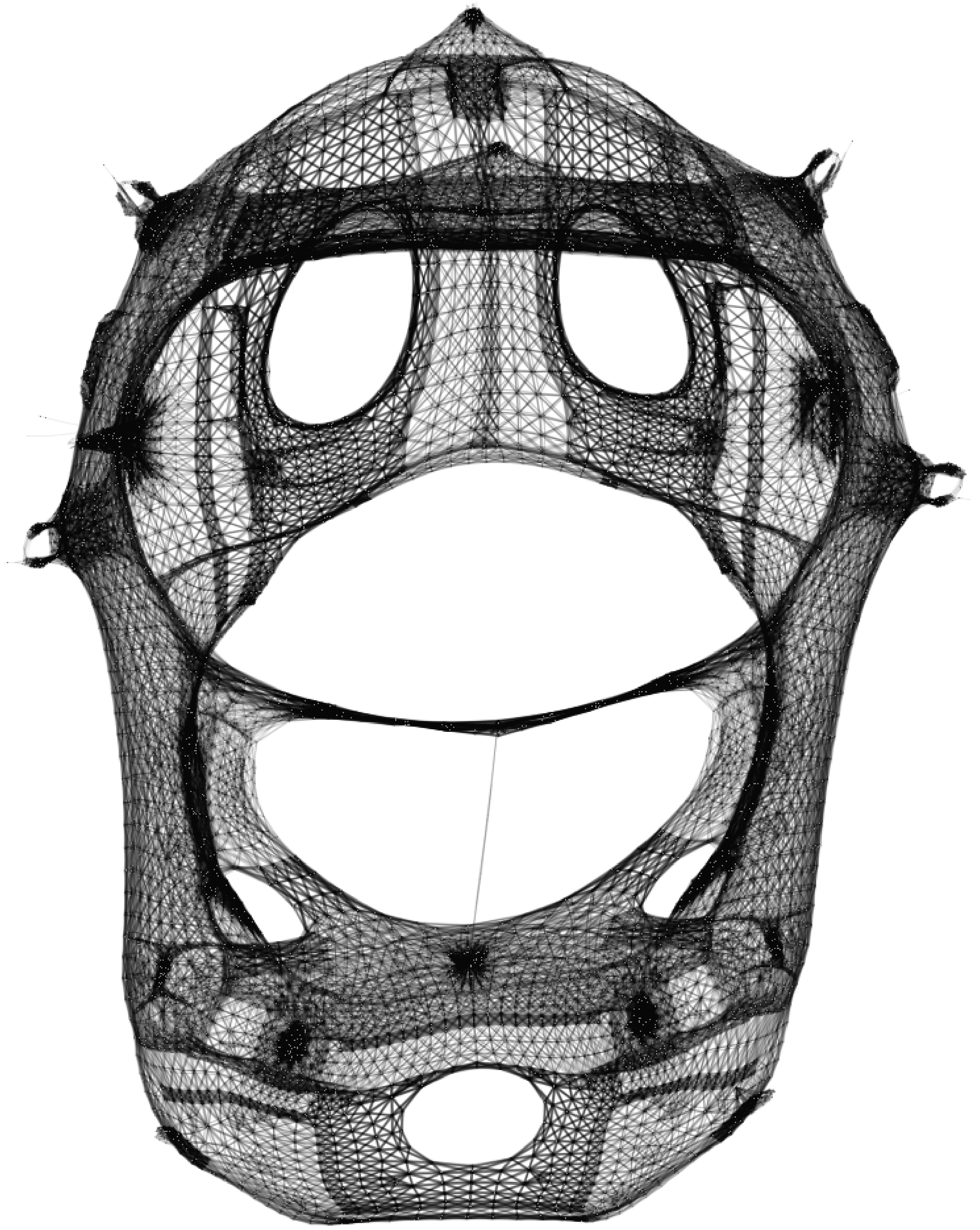


Figure 7.7: Layout of bcsstk31_connected with 35586 nodes and 572913 edges (0.977s).

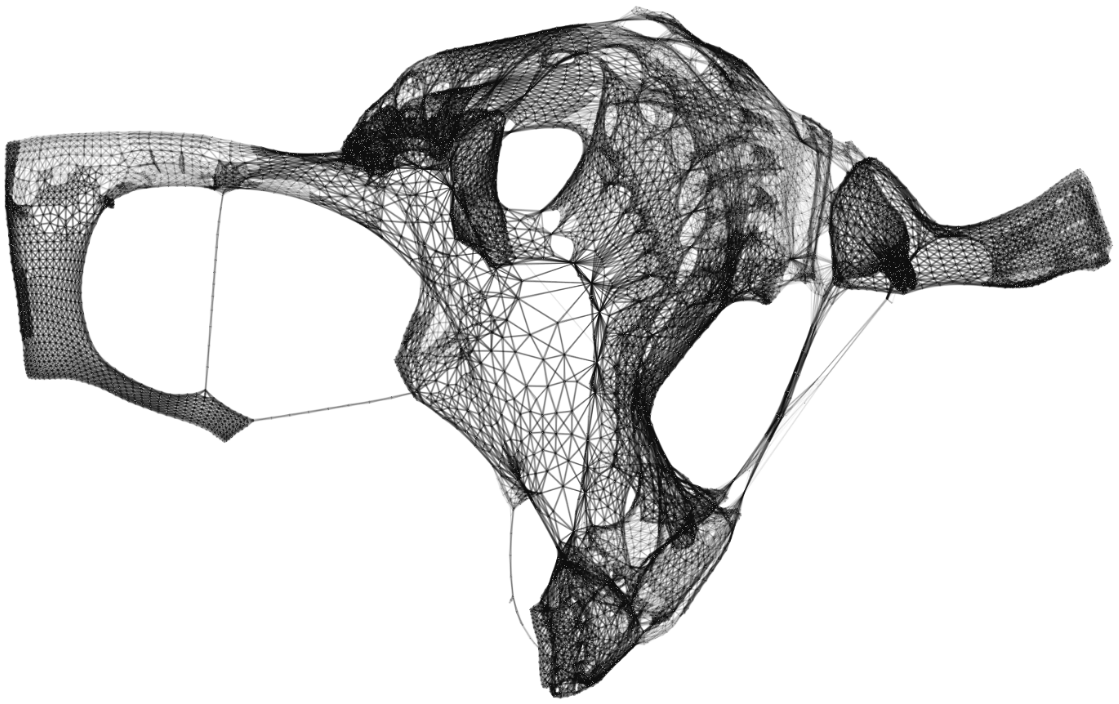


Figure 7.8: Layout of bcsstk32 with 44609 nodes and 985046 edges (1.382s).

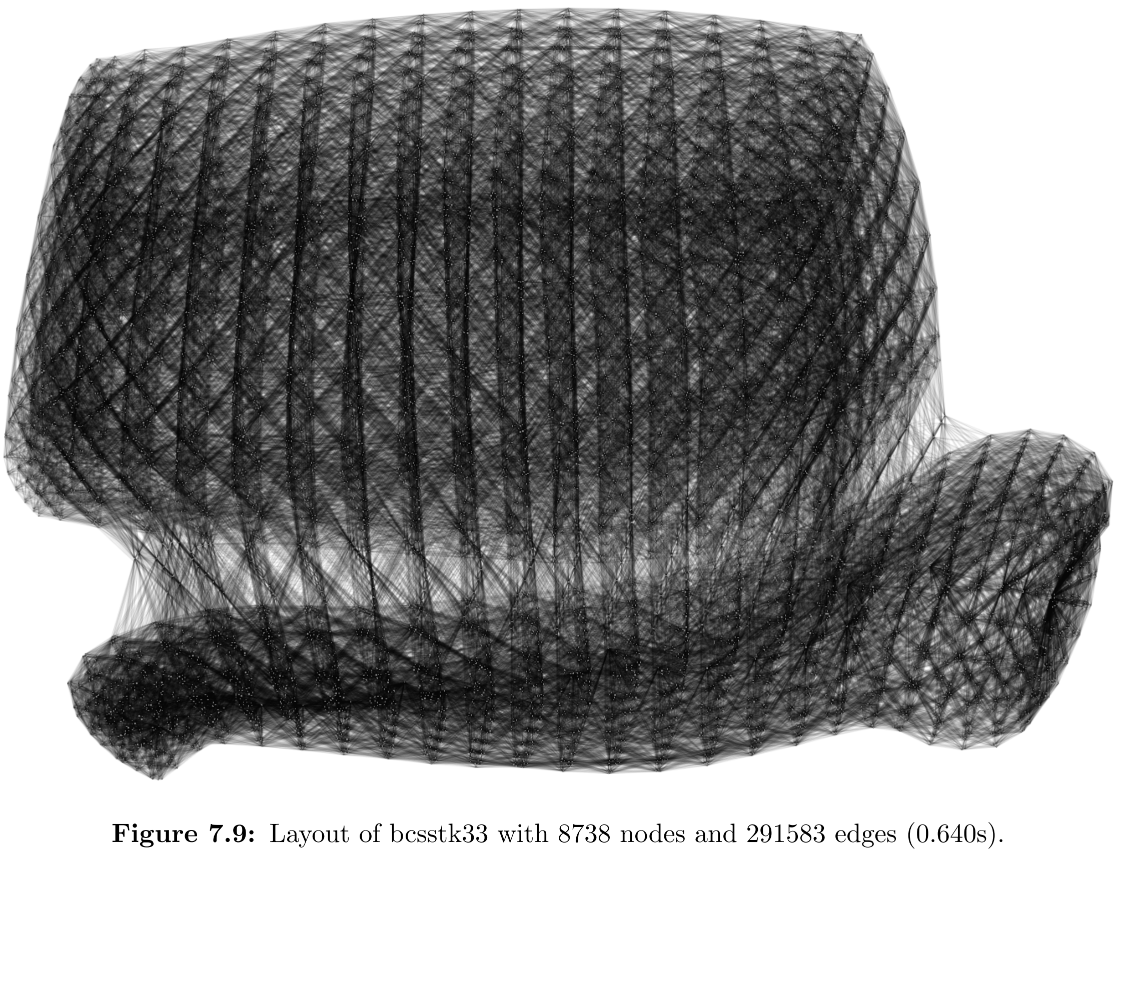


Figure 7.9: Layout of bcsstk33 with 8738 nodes and 291583 edges (0.640s).

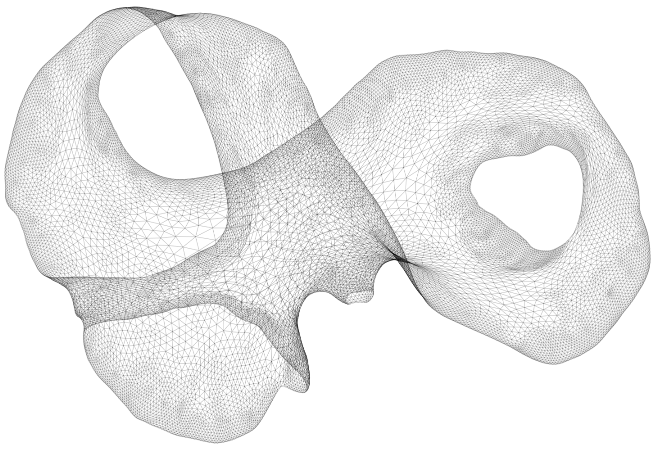


Figure 7.10: Layout of 4elt with 15606 nodes and 45878 edges (0.362s).

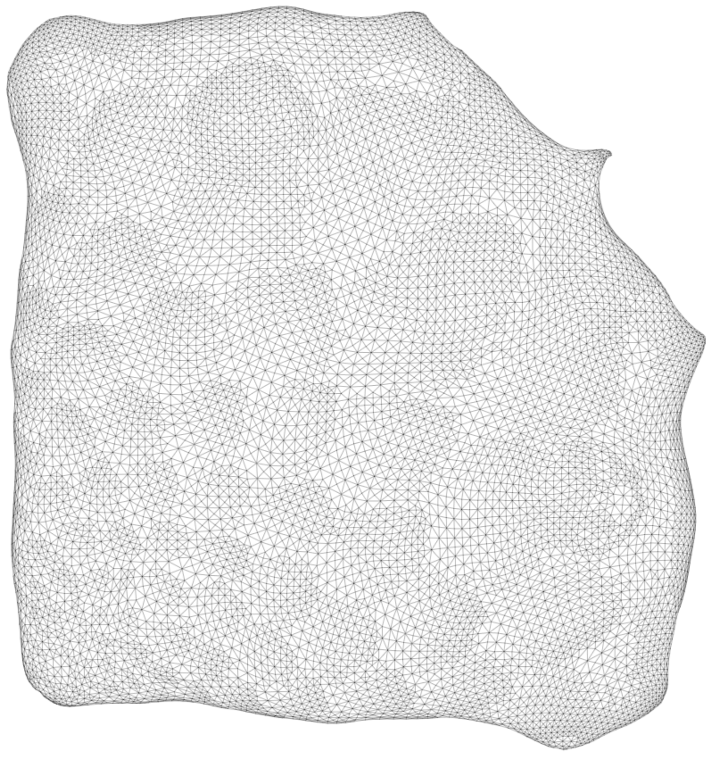


Figure 7.11: Layout of crack with 10240 nodes and 30380 edges (0.365s).

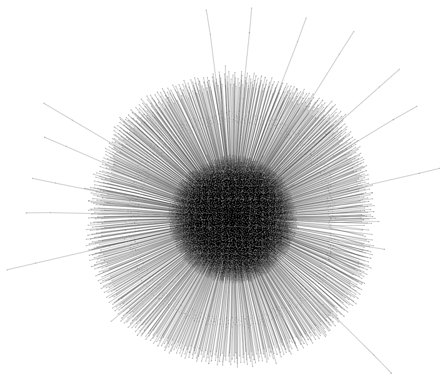


Figure 7.12: Layout of dg_1087 with 7602 nodes and 7601 edges (1.513s).

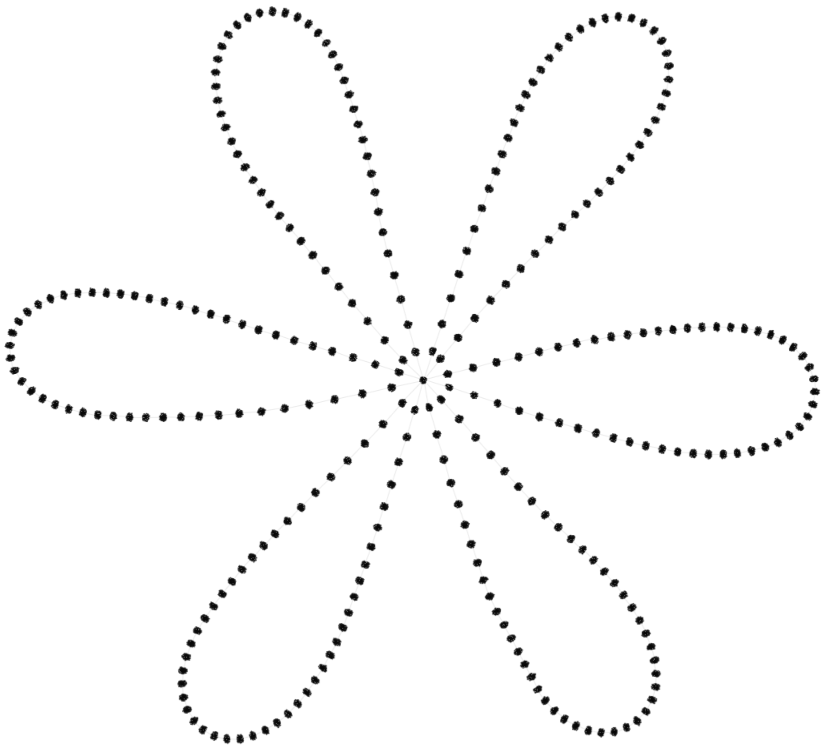


Figure 7.13: Layout of flower_050 with 9030 nodes and 131241 edges (0.279s).

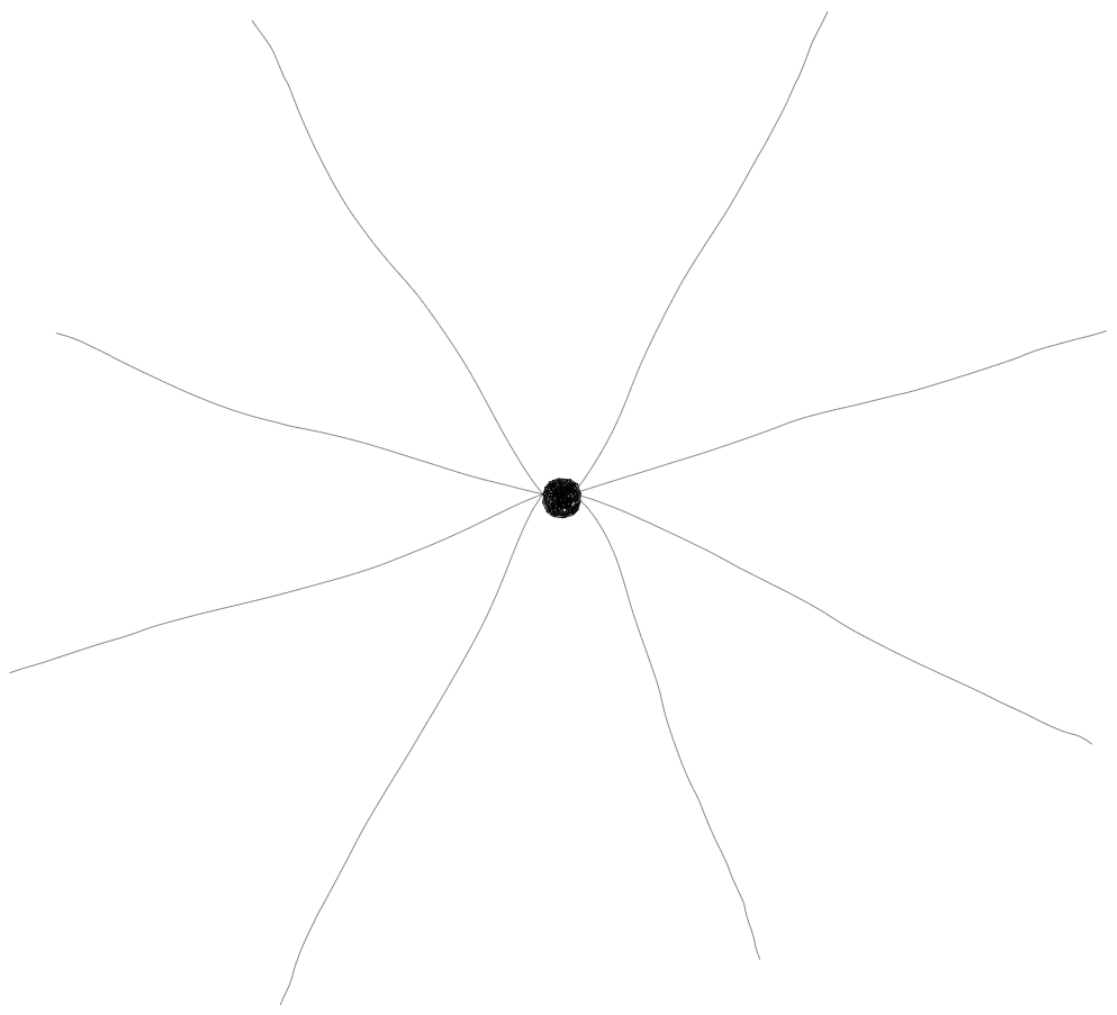


Figure 7.14: Layout of spider_B with 1000 nodes and 1600 edges (0.398s).

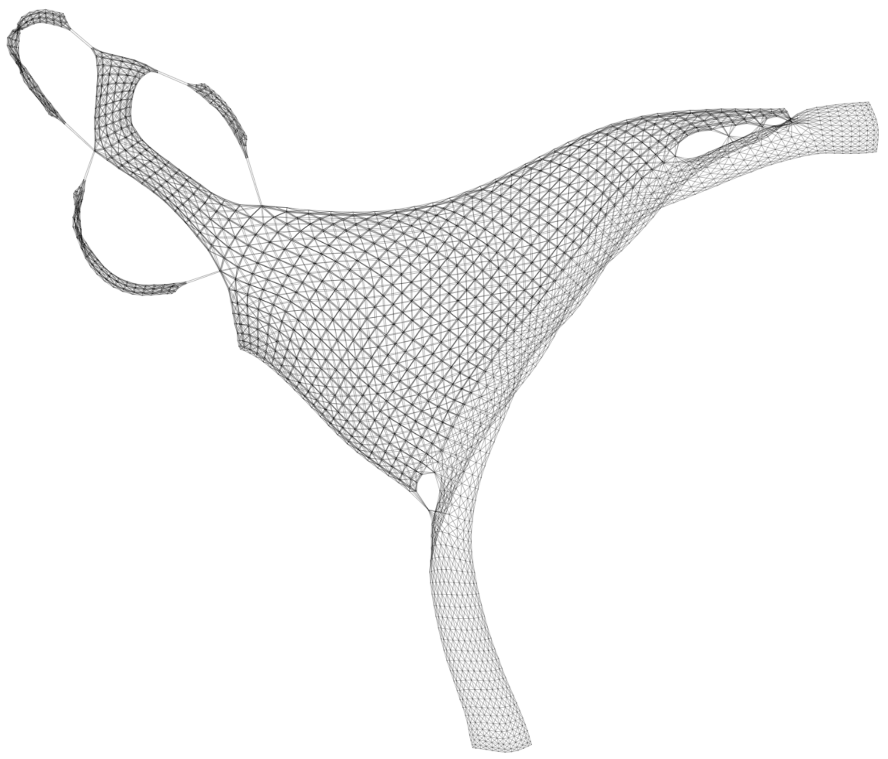


Figure 7.15: Layout of data with 2851 nodes and 15093 edges (0.234s).

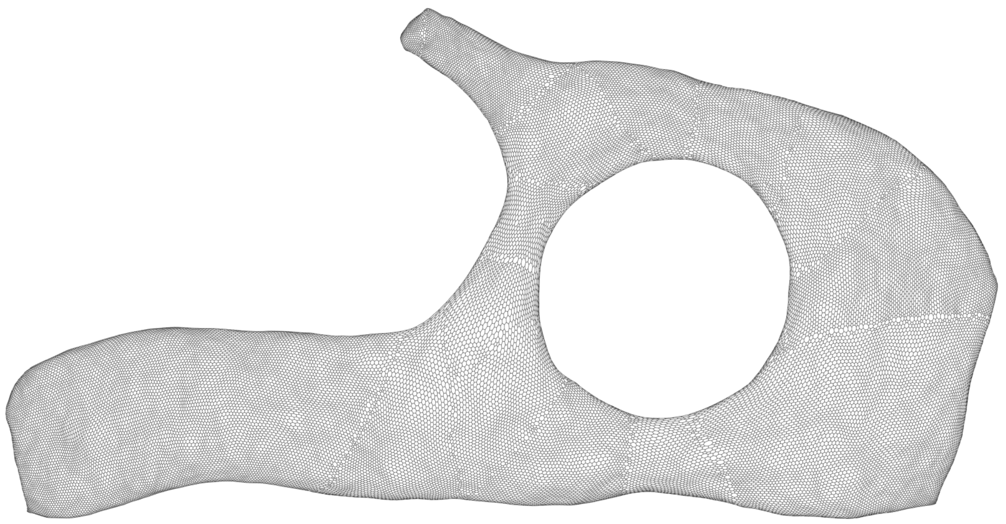


Figure 7.16: Layout of t60k with 60005 nodes and 89440 edges (0.836s).

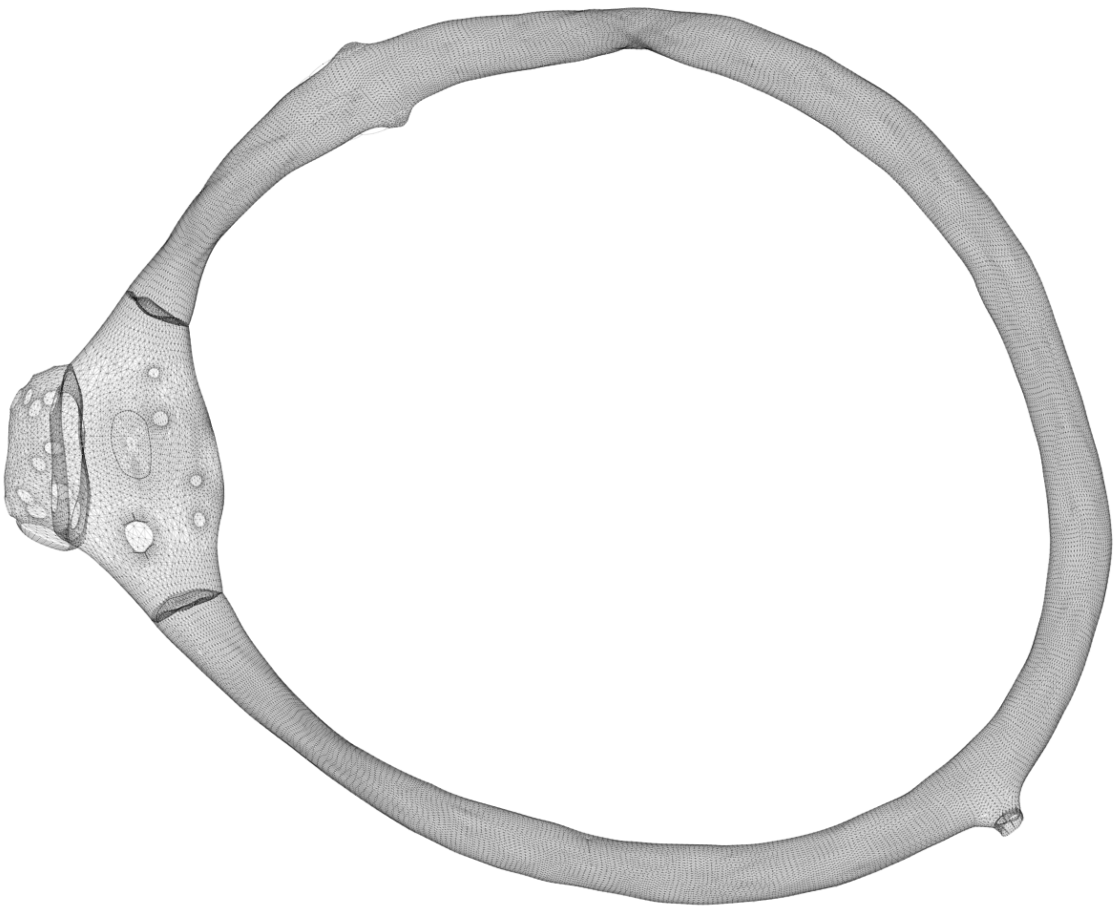


Figure 7.17: Layout of fe_pwt with 36463 nodes and 144794 edges (0.589s).

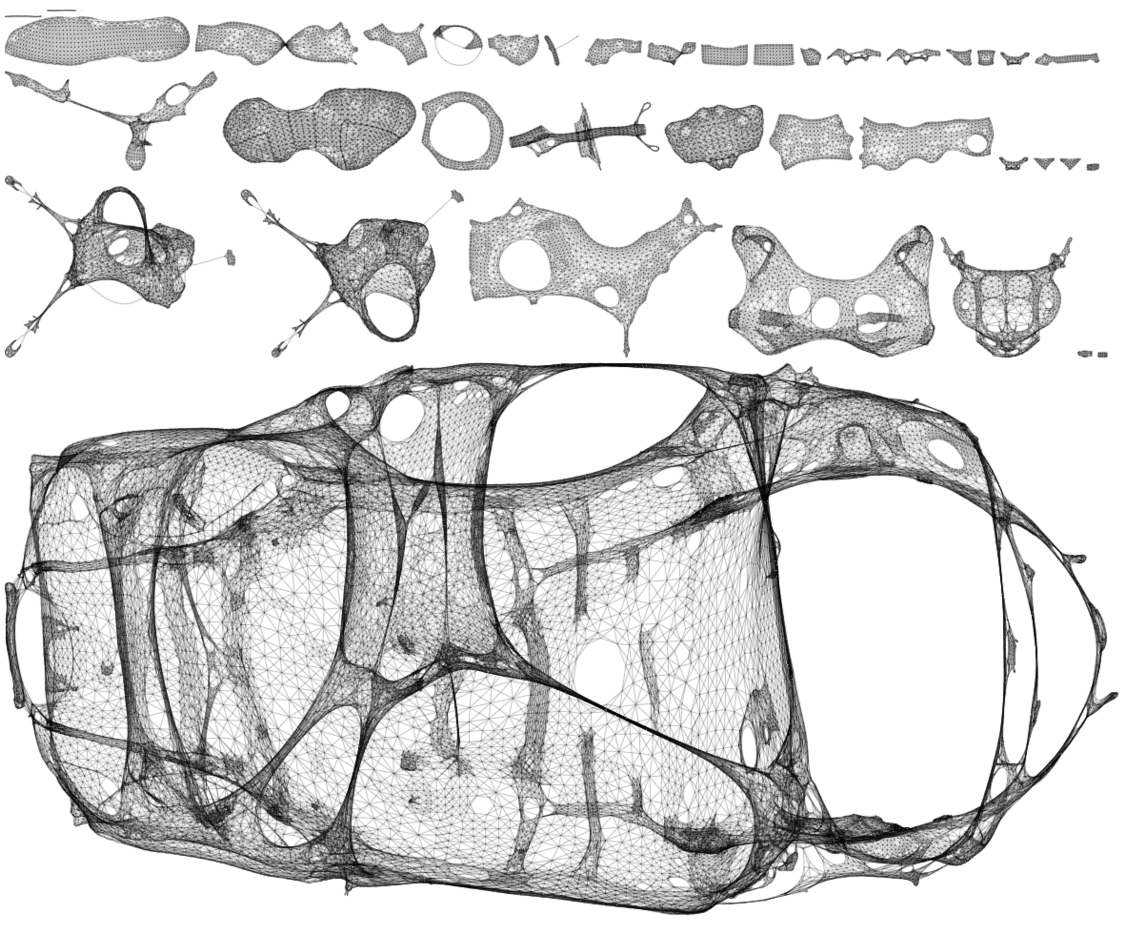


Figure 7.18: Layout of fe_body with 44775 nodes and 163734 edges (4.165s).

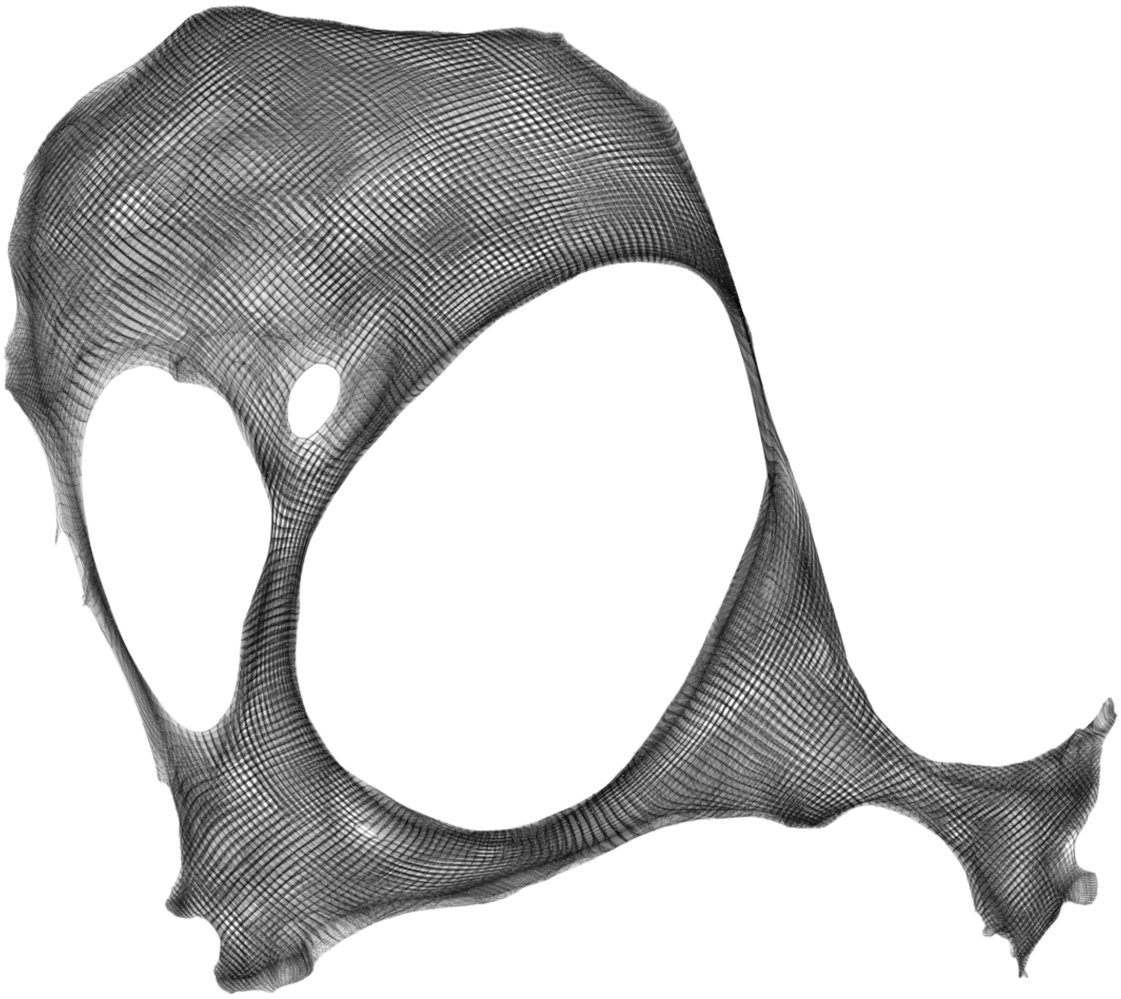


Figure 7.19: Layout of fe_ocean with 143437 nodes and 409593 edges (2.092s).

Chapter 8

Summary and Outlook

In this thesis a new variant of the fast-multipole-multilevel method for drawing large graphs has been developed. The presented algorithm follows only the basic principles of the original method from [Hac05].

We described a new quadtree construction method including a memory layout, the presented method is practically fast and most parts can be easily executed in parallel. The given theoretical runtime of $O(n \log n)$ is domintated by the sorting step, in which we use a standard sorting algorithm. Since, we basically have to sort D-bit integer numbers, one might use a linear time algorithm like radix sort. Furthermore, the algorithm does not depend on anything related to graph drawing, thus it can be easily used for other applications.

This leads to the idea of taking advantage of the computational power provided by modern graphics processors as done in [GHGH09]. The recursive linking function is the only step during the construction process, which prevents the algorithm to run on a modern graphics card. All tools required for the rest of the quadtree construction process already exist, like for example the sorting of the morton numbers.

Furthermore, the presented method is a special version for two dimensions. The principle of the bit interleaving and the computation of the least common ancestor can be easily extended to work in, e. g., three or higher dimensions. The rest of the steps does not have to be modified at all.

We successfully applied the well-separated pair decomposition in combination with the quadtree for the Fast Multipole Method. Which, to our best knowledge, has not been done so far. Again, this approach can easily be modified for higher dimensions.

The multilevel step described in Chapter 6 is not executed in parallel. The reason for this is mainly the high usage of synchronization needed to label or build a graph in parallel. For the future, it remains to find a solution which does not create any parallel edges. Although the removal of parallel edges is fast in practice, the resulting graph fragments the memory since it has to be a dynamic data structure.

List of Figures

List of Algorithms

Bibliography

[APG94] S. Aluru, G. M. Prabhu, and J. Gustafson. Truly distribution-independent algorithms for the n-body problem. In *Supercomputing '94: Proceedings of the 1994 conference on Supercomputing*, 1994.

[BET93] M. Bern, D. Eppstein, and S. Teng. Parallel construction of quadtrees and quality triangulations. In *In Proc. 3rd Workshop Algorithms Data Struct*, pages 188–199. Springer-Verlag, 1993.

[Cal95] P. B. Callahan. *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications.* PhD thesis, John Hopkins University, Baltimore, Maryland, 1995.

[CK95] P. B. Callahan and S. R. Kosaraju. A decomposition of multi-dimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM*, 42:546–556, 1995.

[CwAG99] D. E. Culler and J. P. Singh with A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc, 1999.

[Ead84] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[FR91] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21, 1991.

[Gar82] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, 1982.

[GHGH09] A. Godiyal, J. Hoberock, M. Garlandy, and J. C. Hart. Rapid multipole graph drawing on the gpu. In *Graph Drawing 2009*, volume 5417 of *Lecture Notes in Computer Science*, 2009.

[GR87] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348, 1987.

[Hac05] S. Hachul. *A Potential-Field-Based Multilevel Algorithm for Drawing Large Graphs*. PhD thesis, Institut für Informatik, Universität zu Köln, Germany, `http://kups.ub.uni-koeln.de/volltexte/2005/1409`, 2005.

[HJ04] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Graph Drawing 2004*, volume 3383 of *Lecture Notes in Computer Science*, pages 285 –295, 2004. Extended Abstract.

[HP08] S. Har-Peled. *Geometric Approximation Algorithms (unpublished)*. `http://valis.cs.uiuc.edu/~sariel/teach/notes/aprx/book.pdf`, 2008.

[HRW05] D. Halliday, R. Resnick, and J. Walker. *Fundamentals of Physics*. John Wiley and Sons, 7th edition, 2005.

[Int08a] Intel Coorperation, `http://download.intel.com/design/processor/manuals/253666.pdf`. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 2A: Instruction Set Reference, A-M*, August 2008. version 028.

[Int08b] Intel Coorperation, `http://download.intel.com/design/processor/manuals/253667.pdf`. *Intel 64 and IA-32 Architectures Software Developers Manual Volume 2B: Instruction Set Reference, N-Z*, August 2008. version 028.

[JM04] M. Jünger and P. Mutzel. *Graph Drawing Software*. Springer-Verlag, 2004.

[mil] Millennium simulation project. `http://www.mpa-garching.mpg.de/galform/virgo/millennium/`.

[OGD] *Open Graph Drawing Framework (OGDF)*. `http://www.ogdf.net/`.

[PAC01] H. C. Purchase, J.-A. Adler, and D. Carrington. User preference of graph layout aesthetics: A UML study. In J. Marks, editor, *Graph Drawing 2000*, volume 1984 of *Lecture Notes in Computer Science*, pages 5–18. Springer-Verlag, 2001.

[Pur97] H. C. Purchase. Which aesthetic has the greatest effect on human understanding? In G. Di Battista, editor, *Graph Drawing 1997*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer-Verlag, 1997.

[Tut63] W. T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, 3(13):743–768, 1963.

[Wal] Walshaw graph collection. `http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/`.