Pieces of software for the Coulombic $m$ body problem

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of
Doctor of Technical Sciences

presented by
ROLF STREBEL

born November 29, 1967
citizen of Chur, Switzerland

accepted on the recommendation of
Prof. Dr. René Sperb, examiner
Prof. Dr. Walter Gander, co-examiner

1999
Acknowledgements

I especially thank René Sperb for his interest in Coulomb potentials and for asking my assistance to compute them, otherwise I would probably still search for eigenvalues. We started with an expansion in terms of Bessel functions, but he soon realized that the periodic Coulomb potential can be expressed in terms of exponential functions. Needless to say, this has made the implementation task a lot easier, so you can imagine how grateful I am.

Equally important was the support of Walter Gander, who provided me a pleasant working place in the Institute of Scientific Computing in all these years. The quest for more accuracy in some elementary functions goes back to his initiative. However, the most valuable help, at least in my opinion, was his patience with some of my personal peculiarities. I do like programming, but I do not particularly like typing this thesis, for instance. So thanks for always encouraging me despite this attitude.

The tedious work of reviewing this text was done by Erwin Achermann, Oscar Chinellato, Walter Gander and René Sperb. Thanks for their corrections and suggestions, which have significantly improved the readability. An early draft and ideas for the illustrations were commented on by Erwin Achermann and René Sperb.

This work would not have been possible without other pieces of software, which are often taken for granted. Typically, many people have contributed to a software product, but only few will be mentioned explicitly below. This does not imply that I do not appreciate their work, however. Thanks to the Free Software Foundation for emacs and gcc, to Paul Falstad for zsh, to Richard Stallman and Roland McGrath for make, to Larry Wall for perl, to Matteo Frigo and Steven G. Johnson for fftw, to David H. Bailey and Siddhartha Chatterjee for mpfun++, to Wayne Fullerton for fn, to I. Barrodale and C. Phillips for TOMS/495, to D. E. Amos for TOMS/610, to Tom Rowan for subplex, to Donald E. Knuth for \LaTeX{} and METAFONT, to Leslie Lamport for \LaTeXX{}, to the developers of xfig, to Tomas Rokicki for dvips, to the developers of Maple V, to Cleve Moler and other developers of MATLAB, and to all those I forgot to mention.
Summary

This submitted work describes the efficient computation of the Coulomb potential from different points of view. We show that the efficient implementation even of the simple pairwise computation is nontrivial. An efficient implementation is possible by using simple optimization techniques, which cannot be applied by the compiler alone, however.

Another component of this work is to compute elementary functions like $1/\sqrt{x}$ or $e^x$ quickly and as accurately as possible. Computing $1/\sqrt{x}$ efficiently is helpful for the computation of the pairwise Coulomb interaction, whereas the function $e^x$ is needed for the computation of the exponential expansion.

The main part of our work combines simple optimization techniques and the efficient computation of elementary functions to compute the periodic Coulomb interaction for many particles. This efficient computation is based on an exponential expansion of the Coulomb potential, considering $\cos x$ and $\sin x$ as complex exponential functions for now.

The time complexity of the resulting algorithm for $m$ particles is $O(m \log m)$, which is comparable to other fast algorithms for the same problem, which have time complexities of $O(m)$ or $O(m \log m)$ as well. Since we are ultimately interested in solving a problem as quickly and accurately as possible, the relevant criterion is the true execution time. Consequently, we have put some effort into the analysis of the implementation of the new algorithm and of implementations of other algorithms. Unfortunately, the efficacy of many optimizations is hardware dependent, and it is particularly sensitive on the available processor. We have therefore restricted ourselves to an efficient implementation for the Alpha 21164 microprocessor.

This restriction is admittedly severe, and therefore the efficient implementation is only partly suitable in practice. At least we find that the implementation described in the text is faster than implementations of other algorithms on this particular processor. Furthermore we have shown that we do not have to rely on the compiler alone to optimize a nontrivial algorithm. We think that the non-automatic optimization of programs will play an important role in the future, particularly to bridge the increasing gap between processor and memory speeds.
Zusammenfassung

Die vorliegende Arbeit beschreibt die effiziente Berechnung des Coulomb Potentials aus verschiedenen Gesichtspunkten. Es wird gezeigt, dass schon die effiziente Implementation der einfachen paarweisen Berechnung nicht trivial ist. Eine effiziente Implementation ist bereits mit einfachen Optimierungstechniken möglich, die man allerdings nicht dem Compiler allein überlassen kann.

Eine weitere Komponente dieser Arbeit besteht darin, einige elementare Funktionen wie $1/\sqrt{x}$ oder $e^x$ sowohl schnell als auch möglichst genau zu berechnen. Der Nutzen einer effizient berechneten Funktion $1/\sqrt{x}$ ist für die paarweise Berechnung des Coulomb Potentials ersichtlich, während $e^x$ erst für die Berechnung von Exponentialsummen wichtig ist.

Der Hauptteil der Arbeit kombiniert einfache Optimierungstechniken und die effiziente Berechnung elementarer Funktionen, um das periodische Coulomb Potential und die entsprechenden Kräfte für viele geladene Teilchen zu berechnen. Grundlegend für eine schnelle Berechnung ist die Darstellung des Potentials als Summe von Exponentialfunktionen, wenn man der Einfachheit halber auch $\cos x$ und $\sin x$ als komplexe Exponentialfunktionen betrachtet.

Die Zeitkomplexität des resultierenden Algorithmus für $m$ geladene Teilchen ist $O(m \log m)$, was vergleichbar ist mit anderen schnellen Algorithmen, die für dasselbe Problem Zeitkomplexitäten von $O(m)$ oder ebenfalls $O(m \log m)$ besitzen. Da man letztlich daran interessiert ist, ein Problem möglichst schnell und genau zu lösen, ist allerdings vor allem die reale Berechnungszeit relevant. Wir haben deshalb mit ziemlich viel Aufwand sowohl die Implementation des neu entwickelten als auch die Implementation anderer Algorithmen analysiert. Unglücklicherweise ist die Wirksamkeit vieler Optimierungen abhängig von der Hardware, insbesondere vom vorhandenen Prozessor. Deshalb haben wir uns auf eine effiziente Implementation für den Alpha 21164 Prozessor beschränkt.

D Tools
  D.1 cdup — code duplication  .................................. 145
  D.2 cgen — code generation .................................... 147

E Compilers ...................................................................... 150
  E.1 Timings .............................................................. 150
  E.2 Analysis ............................................................. 151

F Epilogue ......................................................................... 154
  F.1 Alpha 21264 .......................................................... 154
  F.2 Dynamic scheduling ............................................... 155
  F.3 Speculative execution ............................................. 158
  F.4 Preloading ............................................................ 160
  F.5 Conclusion ........................................................... 163
0 Introduction

This text is about software for the Coulombic $m$ body problem. The analysis of this problem and its implementation includes aspects from several areas of computer science and numerical analysis. The main result of this work is a new algorithm MMM\(^1\) to compute the Coulomb interaction for periodic boundary conditions with a time complexity of $O(m \log m)$. We will give a systematic overview of the content after a short summary of how this project evolved.

0.1 History

Initially we were given the problem of computing the Coulomb interaction $\phi$ for charged particles in a box with periodic boundary conditions. Several methods to solve this problem efficiently for a large number of particles already exist, PPPM\(^2\) and FMM\(^3\) for instance. The MMM algorithm proposed in this text uses the Fourier transformation like PPPM, but the transform is done analytically instead of numerically. The MMM algorithm is similar in spirit to FMM, since it evaluates $\phi$ efficiently using the idea of product decomposition. The theory of MMM and a short description of other methods is given in sections 4 and 5. Given this theory, we started implementing the algorithm in a more or less straightforward manner. To our disappointment, performance measurements were not completely satisfactory. In particular, there was a discrepancy between how fast the code could be and how fast the code really was. Attempts to optimize the code by modifications like rearranging innermost loops were only partly successful.

At that point we decided to stick to a particular processor and try optimizations on a lower level. The Alpha 21164 microprocessor was chosen because of its high clock rate and excellent peak performance, which may be exploited by suitably optimized codes. A short description of the Alpha 21164 is given in appendix B. The first experience in manually optimizing an algorithm for the Alpha 21164 was collected in implementing an algorithm to compute $1/\sqrt{x}$ efficiently for several values of $x$. Without the appropriate tools, this experience of writing optimized code in assembly language felt quite laborious, if not painful. However, it turned out that we could express nearly everything in C instead of assembly language, and that a few simple optimization patterns suffice in most cases to obtain reasonable performance. Some optimization techniques are explained in section 1.2 based on notations introduced in section 1.1. The tools used for code generation are described in appendix D. Finally, the algorithm for computing $1/\sqrt{x}$ and its optimization are given in section 2.2. Equipped with this experience and the necessary tools, we were able to implement the overall MMM algorithm efficiently.

0.2 Threads

There are different aspects of the problem the reader may be interested in. To those without particular preferences, we suggest to read the text sequentially. First, some notation and optimization techniques will be introduced in section 1. Then these optimization techniques will be applied to efficiently implement the pairwise Coulombic $m$ body problem in sections 2–3.

\(^1\) MMM $\rightarrow$ (Mean | Messy | Mc | sMc | …) M body Method

\(^2\) PPPM $\rightarrow$ Particle-Particle Particle-Mesh

\(^3\) FMM $\rightarrow$ Fast Multipole Method
The advantage of this problem is its mathematical simplicity, so we may concentrate on the problem of optimization. If you are not scared away by then, you will find a description of the theory and implementation of MMM in the following sections 4–7. Execution times of MMM and comparisons to PPPM and Ewald are given in section 8.

For those with specific interests, we have compiled a list of a few different threads. Below we summarize the contents and describe the sections required to follow particular threads.

**Theory.** This thread is for those interested in the mathematical theory on which MMM is based. Some mathematical preliminaries are given in section 4.1, namely the definition of the Fourier transforms and the definitions and important properties of the modified Bessel functions $K_v$ as well as $\zeta$ and $\psi$ functions. In section 4 we derive a formula for the Coulomb potential for periodic boundary conditions. For distant particles, we essentially obtain an exponential expansion. For particles close to each other, the formula is given in terms of special functions $K_v$, $\zeta$, and $\psi$. The following section 5 summarizes other methods like the Ewald sum, PPPM and FMM. We also explain the idea of product decomposition, using the exponential expansion as an illustrative example.

**Optimization.** This thread describes a few important optimization techniques. Although the techniques themselves are processor independent, their ultimate purpose is to improve the execution time of an algorithm on a particular processor. We have decided to optimize our algorithms for the Alpha 21164 microprocessor, so a short description of important characteristics of the Alpha 21164 is given in appendix B. It is still possible to follow the discussion on optimization techniques in section 1.2 with a general understanding of pipelining and latencies, however. To describe essential features of an algorithm like data dependences and to illustrate its behavior at run time, we introduce some notation in section 1.1, which will be used not only in the explanation of optimization techniques, but through the whole text. One of the primary reasons for this graphical notation is the ability to display algorithms in detail, but avoid to make this text unreadable by tons of displayed code. Optionally, you may look at a simple tool to generate manually optimized code, assuming that the structure of the optimization applied follows a specific pattern. This tool implemented in perl is described in appendix D.

**Elementary.** Based on the optimization techniques described in the optimization thread, we can implement routines to efficiently compute some elementary functions in section 2. For a detailed understanding it is necessary to understand the basics of IEEE floating point numbers described in appendix A. In section 2.2 we describe how to compute $1/\sqrt{x}$ for several values $x$ efficiently. First we explain Goldschmidt’s algorithm and then we implement it using the appropriate optimization patterns. The function $x \mapsto 1/\sqrt{x}$ will be used to compute $1/r$ for given $r^2$ in the pairwise computation of the Coulomb potential. In section 2.4 we describe the computation of pairs $(e^{x}, e^{-x})$, which will be used for terms in the exponential expansion. Not only are these elementary functions important for the overall algorithm, they illustrate the application of several optimization techniques as well.

**Pairwise.** The pairwise computation of the Coulomb interaction is important for several reasons. Since the potential $1/r$ is singular at $r = 0$, approximations for $1/r$ are usually
only valid for \( r \) large enough. For small \( r \), the interaction is computed pairwise in our algorithm. A first possible optimization is to compute \( r^2 \mapsto 1/r \) quickly as described in the elementary thread. Once we may compute \( 1/r \) quickly, the optimization of other parts of the algorithm gains importance, so we will describe and analyze simple ideas for this optimization in section 3.2. In our algorithm it is clear from a partitioning of the box which pairs have to be computed pairwise. In other algorithms, a pair of particles should be considered only if the distance is smaller than some cutoff radius. In section 3.3, we will analyze an approach to efficiently solve the pairwise computation of the Coulomb interaction for a given cutoff radius.

Implementation. The main purpose of the text is the description of the implementation of MMM, so this thread is based on all threads seen so far. In section 6 we continue the discussion from the theory thread and explain the MMM algorithm. For near particles we will need a pairwise computation of the simple \( 1/r \) potential as described in the pairwise thread. The core of the implementation is the computation of the interaction \( \phi \) between distant particles. This contribution can be split into three phases as described in Section 6. Two of these phases are primarily compute bound and are analyzed by conventional techniques in Section 7.2, while the middle phase described in Section 7.3 is memory bound and requires a different type of analysis. To conclude, the overall efficiency of our MMM algorithm is analyzed in Section 8 and compared to the PPPM algorithm for large problems and to the Ewald sum for small problems.

Musings. We have put some effort into making our algorithms run efficiently on the Alpha 21164. A weakness of this approach is that the algorithms will only run efficiently on the Alpha 21164, which is especially bad if we consider that the Alpha 21164 is significantly different from other current microprocessors, even from its successor, the Alpha 21264. Manual optimization is more effective and important for the Alpha 21164 because of its simple implementation, which makes its high clock rate possible. Justifications for manual optimization and a more detailed discussion of similar topics can be found in appendix F, which also describes the idea of preloading and possible implications. The role of compilers and the necessity of manual optimization are discussed briefly in appendix E.

0.3 Innovation

Most of the ideas introduced and explained in this text are not genuinely new, at least not in the sense that no one has thought about them before. Our main contribution is not a particular innovation, but that we committed ourselves to compute the Coulomb potential as quickly as possible. With this goal in mind, we had to solve a diversity of subproblems, which may be classified into mathematics, numerical analysis, algorithms, optimizations or implementation in general, if you like. We mention a few subproblems and their implications on further subproblems.

Elementary functions. Computing \( 1/\sqrt{x} \) quickly is important for the Coulomb potential. To analyze the algorithmic structure, we construct dependence graphs, and we introduce a
graphical issue map to illustrate important run-time characteristics. Classical optimization techniques like loop unrolling and software pipelining are used to obtain an efficient implementation. We have written a simple perl program to actually generate the code of the optimized algorithm.

**Exponential expansion.** The exponential expansion of the periodic Coulomb potential is not entirely new, since it can be derived via simple Fourier transforms. However, the usefulness of this expansion for computing the potential has certainly been underestimated. To make effective use of this expansion, it is necessary to compute $e^x$ and $(\cos x, \sin x)$ quickly. Again, manual optimization and code generation are helpful tools to implement these routines. In addition, the dependence graph can be used to determine reasonable overall structures of their implementation.

**Code analysis.** While issue maps and dependence graphs are useful for compute bound codes, the memory access pattern analysis is useful for memory bound parts of MMM. These two idealizations make it possible to perform execution time analysis on an absolute scale, which means that we compare our implementation to a true optimum, and not just to some other implementation. For memory bound codes, we used another perl tool to manually unroll simple loops.

**Pairwise potential.** It is important to recognize that the pairwise computation of the Coulomb potential can be implemented easily, if speed is not important. There is some potential for optimization both in the computation of pairlists and the computation of potentials and forces. Again, the problems to be solved range from exploiting the memory hierarchy down to using hand-crafted code to avoid branches. Being able to compute the simple pairwise Coulomb potential quickly may even turn out to be the most relevant aspect of our work.

**Efficient implementation.** In the end, we are left with efficient implementations of PPPM and of Ewald’s method for periodic cubic problems, and with an optimized implementation of MMM, which has time complexity $O(m \log m)$ for $m$ particles. It was not clear from the beginning of the project that MMM would be as fast or as complicated as we can see it now. For instance, only the introduction of artificial mirror images reduces the time complexity from $O(m^{7/5})$ to $O(m \log m)$. It is certainly not an exaggeration to say that an important companion during this project was sheer luck.

And now to something completely the same.
1 Basics

In this section we will introduce some notation which will be used throughout the text to analyze code in terms of elementary instructions. This code analysis may then be applied to implement an algorithm efficiently. Consequently, we will explain a few simple but effective optimization techniques below. Applications of these optimization techniques can be seen in later sections. To analyze the efficiency of an implementation, we will measure the execution time on different machines. Most of the time, we are interested in the performance of compute bound codes, so we will use the processor names Alpha, MIPS, UltraSPARC-II, UltraSPARC and SuperSPARC to designate the machines. Technical details are given in appendix C.

1.1 Notation

In this subsection we will first explain the concept of data dependence. Although the concept is simple, information about data dependences for a given piece of code is very important for its analysis and optimization. Often, a compiler cannot precisely determine if there are data dependences, and consequently its optimization may be weak. In contrast, the programmer often has some additional information about data dependences, which may be conveniently expressed with a dependence graph. We will introduce a graphical representation of the dependence graph below.

Some aspects of code analysis like the dependence graph are hardware independent. Since we are ultimately interested in reducing the execution time, most other aspects have to be evaluated in the context of a particular machine. For computationally bound codes, the characteristics of the processor and its floating point unit are important. For memory bound computations, the bandwidth between processor and main memory may be more important. For this reason, we will analyze and optimize our algorithms only for the Alpha 21164 microprocessor, whose basic characteristics are given in Appendix B.

An implementation of a particular algorithm will typically proceed in the following steps. We start with the algorithm itself, perform the code analysis supported by the dependence graph, optimize the code, and finally evaluate its efficiency. To display one aspect of efficiency, the issue map will be introduced to represent the states of functional units during execution.

1.1.1 Data dependences

This subsection contains a brief introduction to data dependences, and introduces some terminology. Basically, there are three types of data dependence, each one illustrated by a short C code sequence below. In the following explanations, the term variable designates the content of a register or a specific memory location.

Read after write or true data dependence. This type of dependence occurs when a specific value is read after it has been written, where the time ordering is implied by the sequential C code. A true data dependence exists for the variable a between the two statements

\[ a = b \times c; \]
\[ d = a + e; \]

It is called true data dependence because there is no way to eliminate this dependence by simply renaming variables.
Write after read or anti-dependence. This dependence occurs when a variable is written after it has been read. For example, there is an anti-dependence for the variable c in

\[
\begin{align*}
    a &= b \cdot c; \\
    c &= d + e;
\end{align*}
\]

This dependence may often be eliminated by renaming c → f in the second statement and the following statements. If the code sequence contains branches, this simple type of renaming is not always possible.

Write after write or output dependence. A variable is written twice, for instance the variable a in

\[
\begin{align*}
    a &= b \cdot c; \\
    a &= d + e;
\end{align*}
\]

In this particular case, the first statement might be eliminated. In more complicated codes however, output dependences may occur without the possibility to eliminate either statement. In some cases, renaming variables may be used to eliminate the dependence by letting a → f in the second and following statements.

Given a sequence of instructions, a data dependence between two instructions generally implies that these two instructions may not be interchanged in the sequence. Often, anti-dependences and output dependences can be eliminated by renaming variables. From now on, we will use the term data dependence or simply dependence for dependences which are essential in the sense that they may not be eliminated in a simple way. For almost all codes to be analyzed this means that we need only consider true data dependences.

1.1.2 Code analysis

There are different aspects of an algorithm, and not all of them are equally important in a particular stage of code analysis. An appropriate representation can emphasize the important aspects, thereby simplifying the analysis. In this subsection, we will introduce several representations or points of view, using a simple example for illustration. By ignoring or neglecting details which are not significant in a certain context, the interesting features will show up more clearly. Of course, the assumption that some aspects of an algorithm may be ignored has to be justified at some point. When counting floating point operations to predict performance, for instance, we must make sure that the cost of memory accesses does not significantly hurt performance.

A first representation is the mathematical description or formula which describes the values we are trying to compute. For example, let us take the very simple formula

\[
r_k = \frac{1}{x_k^2 + y_k^2 + z_k^2} \quad \text{for} \quad 0 \leq k < n
\]

which describes the values \( r_k \) in terms of given values \( x_k, y_k \) and \( z_k \). Note that this is not an algorithm if we are working in finite precision, unless we prescribe an ordering of associative operations and restrict the freedom to apply identities, which would be valid in exact arithmetic.
A second representation is given in explicit algorithmic form by a piece of code in a programming language like C, which introduces other aspects which we did not have to care for in the mathematical formula. First, we have to specify the data layout, i.e., we have to tell the compiler where the data $x_k, \ldots, r_k$ for $0 \leq k < n$ is located. Second, at least in a sequential programming language, we are transforming the formula into a step by step recipe to compute the desired values. The C code for the chosen example might look like

```c
void example (int n, double *x, double *y, double *z, double *r)
{
    int k;
    double xk,yk,zk,tmp;

    for (k = 0; k < n; k++) {
        xk = x[k];
        yk = y[k];
        zk = z[k];
        tmp = xk*xk + yk*yk;
        tmp += zk*zk;
        r[k] = 1/tmp;
    }
}
```

Of course, there are different ways of implementing the computation of a given formula. Although the C code is more detailed than the formula, some obvious facts are not immediately visible any more. It is not obvious any longer that the computations for different values of $k$ are independent, for instance. Furthermore, the sum $x_k^2 + y_k^2 + z_k^2$ may not be considered associative by the compiler. In fact, given no additional information, the compiler will not be able to recognize at compile time that $r_4$ is not the same item as $x_5$, so from the compiler’s point of view there is a possible dependence $r_4 \rightarrow x_5$ and the corresponding instructions may not be interchanged. Consequently, different loop iterations cannot be executed in parallel, unless there is a dependence check at run time. In other cases, the compiler may detect at compile time that no dependence is possible by some clever tests. However, if the compiler is able to detect the absence of dependences, this independence is usually not by coincidence and is probably well known to the programmer. Writing algorithms in C code is often a little verbose, and variable names differ from mathematical symbols in the text. For this reason, we prefer a code representation like the following

```c
function example \equiv
    for k = 0 \ldots n - 1 do
        \langle x, y, z \rangle = \langle x_k, y_k, z_k \rangle \quad \text{— 3 loads}
        t = x_k^2 + y_k^2 \quad \text{— $t$ is $r_k^2$ now}
        t = t + z_k^2
        r_k = 1/t \quad \text{— 1 store}
    od
```

As we have mentioned before, the compiler is often limited in its capability to optimize because it must be conservative with respect to data dependences. If the compiler cannot exclude
the possibility of a dependence, it must assume that there is one. On the other hand, the pro-
grammer often knows about the absence of dependences, although this may be invisible in the code itself. Obviously, with this additional information the code may be optimized more effectively than by the compiler. Consequently, an important piece of information for the man-
ual optimization is the dependence graph, our third representation. Unlike the mathematical
description of the code, the dependence graph does not contain the information required to compute the desired result. The nodes of the dependence graph designate the type of the opera-
tion, but not the specific address of a load, for instance. The target of our manual optimizations
are innermost loops whose iterations may be considered independent. In other words, eventual
dependences between different iterations are structured such a way that they may be ignored.
Therefore, it suffices to look at the dependence graph of a single iteration, which may be given in two variants for our example.

The nodes of the graph denote elementary operations of a particular type, and its edges indicate essential dependences. A list of possible types of nodes is given on the right of the dependence graphs. The reason for having exactly these types of nodes is given in the description of the available pipelines below. The number of operations required is more easily visible in the explicit version, with the disadvantage of an artificial dependence structure. Since operations like addition and multiplication do not allow more than two inputs, only the explicit version corresponds directly to a particular implementation, so we will use the explicit form most of the time.

The fourth representation is the issue map. Like the dependence graph, it does not contain all the information required to compute the result. In some sense, it describes the algorithm at run time by indicating at which clock cycle and functional unit an instruction is issued. The Alpha 21164 is particularly suited for such an approach because of its static scheduling and issue structure. Ignoring variable-length latencies of memory accesses, branches and some exotic instructions, the clock cycles at which a specific instruction is issued can be easily determined. Furthermore, the Alpha 21164 has four fully pipelined functional units, making a two-dimensional display very natural. Consult Appendix B for details about the pipelines and the delays involved for elementary operations. The issue map for our example looks

```plaintext
<table>
<thead>
<tr>
<th>1 2 3 5 ... 10 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock cycle</td>
</tr>
</tbody>
</table>
```
Note that we have attached the types of operations permitted to the corresponding functional unit. Issues of instructions are indicated by filled boxes, where we will often use different shades of gray for different groups of instructions. Note that the division is exceptional insofar as it is not fully pipelined and we need to consider stalls in the divide unit for several cycles after its issue. Since non-pipelined operations will not occur in any of the pieces of code we will analyze, we do not need a particular notion for having a functional unit occupied in the graphics. A weakness of the issue map representation is that it does not take into account widely varying delays of loads depending on the location of the data. Furthermore, a store may result in the store buffer being full, thereby stalling this unit. Therefore, the issue map gives a reasonable impression of the delays between instructions issued, but the optimistic assumption that loads and stores will never lead to stalls are not justified in every case. The issue map is completely misleading for codes which are primarily memory bound, and it is an optimistic representation even for codes with few memory accesses. In contrast, comparisons of the issue maps of non-optimized and optimized codes which you will see later demonstrate very clearly the importance of instruction scheduling to exploit the power of a multi-issue processor.

1.2 Optimization techniques

Essentially, the optimization techniques presented in this subsection are concerned with scheduling a given set of instructions with known dependences. So we assume that other optimizations like common subexpression elimination or strength reduction have already been applied. Since we will schedule our code manually, we restrict ourselves to basic scheduling patterns which are applicable in the context of a loop which is executed many times. The number of iterations necessary to hide the overhead incurred by the optimization is problem dependent, but it is always a good idea to keep this overhead as low as possible so that we have a reasonable performance already for only a few iterations of the loop.

This subsection describes the techniques to optimize loop-structured code using a simple example, which consists solely of floating point multiplications and additions as well as loads and stores. We assume that all loads are from the primary cache and therefore that the load latency is 2 cycles, and that the store buffer may satisfy all requests without stalls. For illustration purposes, we want to compute

\[
   r_k = x_k^2 + y_k^2 + z_k^2 \quad \text{for } 0 \leq k < n.
\]

A possible implementation is given by the following procedure

```
function r ≡
  for k = 0...n - 1 do
    \(\langle x, y, z \rangle = \langle x_k, y_k, z_k \rangle\) — 3 loads
    \(t = x^2 + y^2\) — 2 mul & 1 add
    \(t = t + z^2\) — 1 mul & 1 add
    \(r_k = t\) — 1 store
  od
```
The three scheduling patterns we will explain in this subsection are shortly described below. Note that the term vectorization is not standard, but it expresses the basic idea.

**Vectorization.** This technique commonly called loop distribution splits the loop into two or more parts, and executes the parts in separate loops. This may be useful for large loop bodies which would require too many registers.

**Loop unrolling.** This technique combines several iterations of the loop into one basic block. This is useful to reduce the cost of loop overhead for small loop bodies and to increase the potential for instruction-level parallelism.

**Software pipelining.** Overlap different iterations of the loop in a particular way. This may be useful to increase the potential for instruction-level parallelism.

Note that in practice these methods may be combined in some way or other, for instance by unrolling a loop and then software pipelining the resulting code. Furthermore, there is often some potential for unstructured scheduling, which operates on the instruction level and does not follow a specific pattern. Applications of these methods can be seen in later sections. For another description of optimization techniques, partly within the context of hardware architecture, consult [19] and the references therein.

### 1.2.1 Vectorization

The technique of vectorization is best explained using the example given above. For simplicity, the loop is split into two parts, but more general splits are possible.

```plaintext
function r_v ≡
for k = 0...n - 1 do — first part
   (x, y) = (x_k, y_k)
   t = x^2 + y^2
   t_k = t — scalar → array
od
for k = 0...n - 1 do — second part
   t = t_k — scalar ← array
   z = z_k
   t = t + z^2
   r_k = t
od
```

The two loops communicate necessary scalar data, in our case the variable t, through an array \( t_k \). This promotion of a scalar variable to an array to reduce artificial output dependencies is typical for vectorized code. Of course, writing the two loops sequentially in the code does not necessarily imply that the loops have to be executed one after the other, but the \( k \)-th iteration of the second loop may start as soon as the \( k \)-th iteration of the first loop has finished. In hardware, this technique of overlapping a computation on vectors with a data dependence is called chaining. In principle, chaining is just a particular pattern of instruction scheduling, and corresponds to software pipelining in software terminology.
In a dependence graph, the above transformation may be sketched as

where we have introduced a notation for having many copies of the same dependence graph, with no data dependences between two iterations. For our example, vectorization alone is not enough to exhibit enough instruction-level parallelism, since some data dependences within an iteration remain. But in conjunction with other techniques, vectorization may still be useful, since it reduces the number of registers required or transforms a complicated loop into a simpler loops, thus making manual optimization more manageable.

1.2.2 Loop unrolling

The technique of loop unrolling transforms a loop iterating over single instances of the loop body into another loop iterating over groups of loop bodies. If different instances of the loop body are independent, loop unrolling corresponds to executing several bodies in parallel, and simply iterating over this parallelized bit. Of course, the joined bodies may further be optimized by general instruction scheduling techniques. Using the issue map, the code transformation by loop unrolling can be easily explained. But since this is our first application of the issue map, let us start with the code where all instructions are explicitly placed according to their issue cycle. Assuming that loads and stores are no problem, the code looks as follows.

```c
function r =
loop :
    x = x_k; y = y_k
    z = z_k, k = k + 1
    x = x^2, x_k \rightarrow x_{k+1}, y_k \rightarrow y_{k+1}
    y = y^2, z_k \rightarrow z_{k+1}, r_k \rightarrow r_{k+1}
    z = z^2, cont = k < n
    t = x + y
    t = t + z
    r_k = t, if cont goto loop
```
Note that we had to invent some extra notation $x_k \rightarrow x_{k+1}$ to indicate the pointer arithmetic required. This is a consequence of the fact that this stylized code does not allow the programmer to express low level operations naturally. On the other hand, it is one of the strengths of the programming language C that low level operations may be expressed easily most of the time. At least for common elementary operations, it is possible to write C code in such a way that every C statement translates to exactly one machine instruction. For our example, the C equivalent of the loop would look

```c
loop:
  xk = x[0]; yk = y[0];
  zk = z[0]; k++;
  xtmp = xk*xk; x++; y++;
  ytmp = yk*yk; z++; r++;
  ztmp = zk*zk; cont = (k < n);
  /* delay 2 cycles */
  tmp = xtmp + ytmp;
  /* delay 3 cycles */
  tmp = tmp + ztmp;
  /* delay 3 cycles */
  r[-1] = tmp; if (cont) goto loop;
```

The representation of this code in the issue map is obtained by looking only at the operation type and forget about operands. There is some more information we can put into the issue map. In our example, some instructions belong to the proper algorithm, while others like incrementing $k$, adjusting pointers or the branch fall into the category of loop code. The issue map given below colors original instructions gray and loop instructions black.

Additionally, we have put a symbol to indicate the number of iterations required, a notation which will be useful to explain the idea of loop unrolling.

The concept of loop unrolling consists of three phases, which are illustrated in the following graphic. Note that we unroll the loop threefold in this example, so we assume for simplicity that $n$ is a multiple of 3. For our example chosen, there are no essential dependences between different iterations of the the loop body. This property simplifies the instruction scheduling after joining the bodies belonging to different iterations, but the overall process is the same for more general codes.
Note that different instances of the loop body are distinguished by shades of gray, while the loop code is colored black. Usually, the execution time is improved for two reasons:

(i) Some of the loop overhead can be eliminated, both in terms of instructions and branches required. Although not shown here, there is often some penalty involved for taken branches, even if they are correctly predicted. The number of branch mispredicts usually remains unchanged with loop unrolling, however.

(ii) There are more instructions to schedule in each loop body, often from independent iterations, which is a particularly easy case. Of course, the new loop body will always be at least as long as the original one, but the number of iterations is only a fraction of the original number.

On the other hand, loop unrolling implies increased code size, which may hurt performance. Furthermore, if $n$ is not known to be a multiple of 3, we need another piece of code—often the original slow code—iterating $r$ times, where $r$ is the remainder of $n$ divided by 3. Typically, loop unrolling is beneficial in terms of execution time if the number of iterations is not too small and the loop body is not too large.

### 1.2.3 Software pipelining

The technique of software pipelining is a way to overlap the execution of different instances of the loop body in a systematic way. Software pipelining primarily helps to hide latencies due to data dependences within a single iteration. Unlike with loop unrolling, the loop overhead is not reduced, so the loop overhead should be relatively small for this technique to make sense. In practice, the two techniques are often combined by first unrolling the loop—and therefore increasing the size of the loop body to some extent—and then software pipelining the resulting code. We will see this technique applied for several problems in later sections. A graphical illustration of the technique of software pipelining applied to our example is given below in terms of issue maps. In this specific case, we have overlapped two iterations, and the loop instructions are colored black again.
Of course, more than two iterations may be overlapped in general. Correspondingly, the pre-
and post-phases grow and the inner loop will be executed \( n + 1 - k \) times for \( k \) overlapped
iterations. As can be seen from the graphic, the loop body may become shorter thanks to our
freedom to schedule instructions from different iterations, but the loop overhead stays more or
less the same.

In case you do not feel comfortable yet with the issue map, here is the code corresponding to
the end result of the above transformation, assuming that \( n \geq 2 \). If nothing is known about
\( n \), we may use a separate piece of code for small \( n \). In simple cases like the piece of code
below, additional branches to check for \( n = 0 \) and \( n = 1 \) may be inserted, thus avoiding
this unwanted code duplication. Again, each line of code corresponds to one cycle, except for
labels and delays.

```plaintext
function r =
    x = x0, y = y0
z = z0, k = 1
x = x^2
y = y^2
z' = z^2

loop :
x = x_k, y = y_k
z = z_k, k = k + 1
x = x^2, x_k \rightarrow x_{k+1}, y_k \rightarrow y_{k+1}
y = y^2, t' = t + z', z_k \rightarrow z_{k+1}, r_k \rightarrow r_{k+1}
z' = z^2, cont = k < n

2x delay
```

---

© 2023. All rights reserved.
\[ t = x + y, r_{k-2} = t', \text{ if } \text{cont goto loop} \]
\[ t' = t + z' \]
\[ r_{k-2} = t' \]

Note that additional variables \( z' \) and \( t' \) had to be introduced since the formerly used \( z \) and \( t \) are overwritten by an operation from a later iteration. The problem of variables being reused and potentially being overwritten is quite common when applying software pipelining. Often it helps not to reuse the same variable for several values as we did in the original code with \( t \) by having \( t = x + y \) and \( t = t + z \) instructions. In other cases, it is necessary to copy, reload or even recompute a value which we want to keep for longer than one loop.

We have already mentioned that we may use the same code for \( n = 1 \) and \( n > 1 \) iterations for the 2x pipelined case by inserting a conditional branch. Symbolically, this can be done as follows

\[ \begin{array}{c|c|c|c}
\text{iteration} & n = 1 & n > 1 & \text{pipelined} \\
\hline
n & 1 & 2 & 2 \\
\hline
\end{array} \]

In the general case of \( k \times \) pipelining we may avoid code duplication as well for the special cases \( n < k \). The graphic below shows how to structure the phases for the case \( k = 3 \), but the same idea works for arbitrary \( k \) as well.

\[ \begin{array}{c|c|c|c|c|c|c|c|c}
\text{iteration} & n = 1 & n = 2 & n > 2 & \text{pipelined} \\
\hline
n & 1 & 2 & 2 & 1 & 3 & 2 & 1 & 3 & 2 & 3 \\
\hline
\end{array} \]
2 Elementary functions

In this section we describe how to efficiently compute particular elementary functions \( f(x) \) for a vector of input arguments \( x \). We are primarily interested in computing \( 1/\sqrt{x} \) for the pairwise Coulombic potential and pairs \((e^x, e^{-x})\) for the exponential expansion, as we will see later. Though not equally important, we are further interested in computing pairs \((\cos x, \sin x)\). Out of curiosity, and because it is relatively easy to do, we will compute \( 1/x \) as well.

It is not immediately obvious that we can compute one of these \( f(x) \) significantly faster than optimized library routines or even hardware, if we consider the reciprocal \( f(x) = 1/x \). However, it is possible for the following reasons. First, we exploit the fact that we compute \( f(x) \) for several \( x \) independently. Second, hardware division is relatively slow, and the efficiency of library routines is limited, since they have to solve the scalar problem. Third, we may compute \( z = 1/\sqrt{x} \) in one step, compared to computing \( y = \sqrt{x} \) first by calling a library routine and \( z = 1/y \) by hardware division. Similarly, we may compute the pair \((y, z) = (e^x, e^{-x})\) at the same time, instead of computing \( y = e^x \) and then \( z = 1/y \).

One price we will pay for the increased efficiency is that the optimized routine to compute \( f(x) \) will not work correctly for all input arguments \( x \). For example, the function \( 1/\sqrt{x} \) expects that \( x > 0 \) and that \( x \) is a normalized floating point number. The conditions imposed by other functions \( f(x) \) are given in the specific subsections. Since accuracy is not always equally important, we have implemented two routines for the case \( 1/\sqrt{x} \), the emphasis being either more on efficiency or on accuracy.

After a detailed analysis of the efficient implementation of the functions \( 1/\sqrt{x}, 1/x, (e^x, e^{-x}) \) and \((\cos x, \sin x)\) we will conclude this section by discussing advantages and disadvantages of this approach. But first we will introduce some notation and an execution time model common to all functions \( f(x) \).

2.1 Introduction

First, this subsection introduces some notation which is used to explain operations on the floating point representation. Then we define some error measures we will use to assess the accuracy of the algorithms. Finally, we introduce a simple model for the execution time of an optimized routine depending on the length of the input vector.

2.1.1 Floating point

In the following subsections we will be concerned with computing functions \( f(x) \) in double precision format. Since we will manipulate the floating point representation directly, you should be acquainted with the floating point format. A short introduction to the IEEE 754 floating point standard is given in Appendix A.

For a floating point number \( x \), we use the notation \( x = (s|e|m) \) to designate the sign bit \( s \), the unbiased exponent \( e \), and the fraction \( m \). Note that we will usually ignore the bias in the exponent and work with the unbiased exponent \( e \). Assuming that \( x \) is normalized, we may write

\[
x = (-1)^s \times 2^e \times (1.m),
\]
where \((1.m)\) is the mantissa of \(x\). The notation \((\ast|\ast)\) represents the concatenation of fields. The inverse operation, field extraction, is best explained by the following example. To extract the biased exponent \(\bar{e}\) of a floating point number \(x\), we could use \((\ast|\bar{e}|\ast) = (\bar{x})\) or more explicitly \(\bar{e} = (x)_{62..53}\). The former expression may only be used if the width and position of \(\bar{e}\) are clear from the context.

2.1.2 Error measures

Using the notation from above, the double precision floating point number \(x = 1\) in terms of its fields is \(x = (0|0|0 \ldots 0)\). The next larger floating point number \(x^+\) is then given by \(x^+ = (0|0|0 \ldots 01)\). The difference \(\epsilon_m = x^+ - x = 2^{-52}\) is called machine precision, and approximately we have \(\epsilon_m \approx 2.22 \times 10^{-16}\).

Assume that we want to compare a numerical result \(z\) to the exact result \(\bar{z}\). Then the above suggests the following error measure

\[
\epsilon_{ulp} = \frac{z - \bar{z}}{2^e - \epsilon_m},
\]

which is equivalent to \(\epsilon_{ulp} = (z - \bar{z}) \times 2^{-e}\) for \(z = (-1)^e \times 2^e \times (1.m)\). The quotient \(\epsilon_{ulp}/\epsilon_m\) measures the error in multiples of the least significant digit of \(z\). Consequently, an error \(|\epsilon_{ulp}| < \epsilon_m/2\) means that \(z\) is the floating point number closest to \(\bar{z}\), except for borderline cases like \(z = 1\) and \(\bar{z} < 1\).

While \(\epsilon_{ulp}\) depends on the internal representation of floating point numbers, the absolute and relative errors

\[
\epsilon_{abs} = \frac{z - \bar{z}}{} \quad \text{and} \quad \epsilon_{rel} = \frac{z - \bar{z}}{|\bar{z}|},
\]

are defined independently of the floating point format. From the definitions we see that \(\epsilon_{abs} = 2^e \times \epsilon_{ulp}\) for \(2^e \leq |z| < 2^{e+1}\). Consequently, a numerical result \(z\) may be considered optimal if \(|\epsilon_{abs}| \leq 2^{e-1} \times \epsilon_m\).

The measures \(\epsilon_{ulp}, \epsilon_{abs}\) and \(\epsilon_{rel}\) indicate the error for one pair \((z, \bar{z})\) of computed and exact value. If we are given several pairs \((z_k, \bar{z}_k)\) and therefore a sequence \((\epsilon_k)_{k=1}^n\) of errors, it makes sense to summarize the overall error. Using a standard statistical approach, we will represent the overall error by the average value \(\bar{\epsilon} = \mu_\epsilon\) and the standard deviation \(\sigma_\epsilon\) of the sequence. By definition

\[
\bar{\epsilon} = \mu_\epsilon = \frac{1}{n} \sum_{k} \epsilon_k \quad \text{and} \quad \sigma_\epsilon^2 = \frac{1}{n-1} \sum_{k} (\epsilon_k - \bar{\epsilon})^2.
\]

We will use the notation \(\epsilon = \mu_\epsilon \pm \sigma_\epsilon\) for this overall error of the sequence \((\epsilon_k)\) in the statistical sense. If \(\mu_\epsilon \approx 0\), then the standard deviation \(\sigma_\epsilon\) may be regarded as a typical error of the sequence.

2.1.3 Execution time

The optimized routines to compute one of the functions \(f(x)\) expect a vector of input arguments \(x\) and will return a vector of results \(z\). Ideally, the execution time \(t\) required per element would
be independent of the vector size $n$. However, the function call overhead alone implies that $t$ will decrease for increasing $n$, since the one-time overhead is now shared by $n$ elements.

A more realistic approach is the following idealized model for the execution time $t$ per element

$$ t = t_0 + c_1 + c_2/n, $$

where $t_0$ is the minimal time required, which may be easily derived from the issue map. The other parameters $c_1$ and $c_2$ may be found by experiment. In this idealized model, the parameter $c_1$ will be mainly determined by the overhead which occurs for every iteration of the innermost loop, a taken branch penalty for instance. In contrast to this, the parameter $c_2$ measures the overhead which occurs once per function call or once per vector. Part of the overhead $c_2$ are for example function call and return, register save and restore, and the penalty due to non-optimal pre-loop and post-loop codes. For large vectors and therefore $n \to \infty$, the overhead $c_2$ will be negligible and the execution time will be $t \to t_0 + c_1$. However, not every piece of overhead can be easily classified to either class $c_1$ or $c_2$. For example, load misses due to table lookup may occur in every iteration for small $n$, but for larger $n$ table entries will be reused and the number of load misses will decrease.

To find the overheads $c_1$ and $c_2$ for a particular function $f(x)$, we will measure the execution time $t$ per element for varying vector length $n$. Then the parameters $c_1$ and $c_2$ can be determined by fitting the measured data to the simple model in the least squares sense, minimizing the relative errors between predicted and measured execution times.

### 2.2 Computing $1/\sqrt{x}$

In this subsection we describe how to efficiently compute $1/\sqrt{x}$ for a vector of $x$ values. The idea to compute $1/\sqrt{x}$ quickly in software to speed up the computation of Coulombic potentials is not new, see [15] for instance. Since accuracy is not always equally important, we have implemented two routines to compute $1/\sqrt{x}$, the emphasis being either more on efficiency or on accuracy. The faster and less accurate algorithm is based on Goldschmidt's idea, the accurate algorithm uses the binomial expansion. A description of Goldschmidt's idea and an introduction to computer arithmetic in general can be found in [10]. First, we will describe the algorithms and analyze their accuracy. Then, we display the optimization techniques applied and finally analyze the efficiency of the implementation.

#### 2.2.1 Algorithm

In this subsection we will describe two algorithms to compute $1/\sqrt{x}$, the more accurate one being based on the binomial series. The main difference between these algorithms is the way of computing $1/\sqrt{x}$ for $x \approx 1$, while the argument reduction from an arbitrary $x$ to $x \approx 1$ is similar.

Internally, floating point numbers are stored as tuples $<s|e|m>$ corresponding to the value $x = (-1)^s \times 2^e \times (1.m)$, if we ignore the bias in the exponent $e$. Since $1/\sqrt{x}$ is undefined for $x < 0$ we will assume from now on that $s = 0$. For $e = 2e' + e''$ with $e' = [e/2]$ we may then write

$$ 1/\sqrt{x} = 2^{-e'} \times 1/\sqrt{x_m} \quad \text{with} \quad x_m = 2^{e''} \times (1.m), $$
where \( e' \) and \( e'' \) are integral. The multiplication with \( 2^{-e'} \) is easy in the floating point format, since we may simply add \(-e'\) to the exponent field. We have therefore reduced the problem of computing \( \frac{1}{\sqrt{x}} \) for arbitrary \( x \) to the interval \( 1 \leq x_m < 4 \), since \( e'' = 0 \) or \( e'' = 1 \). To further reduce the argument \( x_m \) we find an approximation

\[
x_t = 2^{e''} \times (1.t|1|0\ldots0) \approx x_m = 2^{e''} \times (1.m)
\]

where \( t \) contains the first few bits of \( m \), so that \( y_t = \frac{1}{\sqrt{x_t}} \) may be determined by table lookup with index \( (e''|t) \), which is \( b \) bits wide. Note that we use \( (1.t|1|0\ldots0) \) to approximate \( (1.m) \) with a maximum absolute error of \( 2^{-b} \) for \( t \) being \( b - 1 \) bits wide, compared to a maximum error of \( 2^{-b+1} \) for another approximation \( (1.t|0\ldots0) \). Assuming that we have determined \( e', x_t = \frac{1}{\sqrt{x_t}} \) and \( x_s = x_m - x_t \) we may write

\[
\frac{1}{\sqrt{x}} = 2^{-e'} \times \frac{1}{\sqrt{x_t}} \times \frac{1}{\sqrt{x_m/x_t}} = 2^{-e'} \times y_t \times (x_m y_t^2)^{-1/2} = 2^{-e'} \times y_t \times (1 + x_s y_t^2)^{-1/2}.
\]

Since \( x_t \approx x_m \) we are essentially left with the problem of computing \( \frac{1}{\sqrt{u}} \) for \( u = x_m y_t^2 \approx 1 \). Both algorithms we will describe below will make use of this argument reduction. The so-called Goldschmidt’s algorithm to compute \( \frac{1}{\sqrt{x}} \) is very efficient, but not entirely accurate. In contrast to this, an approach using the binomial series is a little slower, but the accuracy attained is close to machine precision.

**Goldschmidt’s algorithm**

The simple idea of Goldschmidt’s algorithm is to find a sequence

\[
x = x_0 = x_1 = \ldots = x_k = \ldots
\]

such that \( x_k \to 1 \). It is easy to see that \( x = x_\infty / y_\infty \) and \( x_\infty = 1 \) imply that \( y_\infty = 1 / \sqrt{x} \). To let \( x_k \to 1 \), we write the quotient for the step \( k \to k + 1 \) as follows

\[
\frac{x_k}{y_k} = \frac{x_k r_k^2}{(y_k r_k^2)^2} = \frac{x_{k+1}}{y_{k+1}^2},
\]

where \( r_k \approx 1 / \sqrt{x_k} \). Assuming that \( x_k \approx 1 \) for all \( k \), we may write \( x_k = 1 + \delta_k \). Then a good approximation to \( 1 / \sqrt{x_k} \) is given by \( r_k = 1 - \delta_k / 2 = 3 / 2 - x_k / 2 \). With this choice of \( r_k \), the step \( x_k \to x_{k+1} \) may be written

\[
x_{k+1} = x_k r_k^2 = (1 + \delta_k) (1 - \delta_k / 2)^2 = 1 - \frac{3}{4} \delta_k^2 + \frac{1}{4} \delta_k^3 =: 1 + \delta_{k+1},
\]

so \( (x_k) \) converges quadratically to 1. To obtain a \( x_0 \approx 1 \) for arbitrary \( x \), we use the approximation from the argument reduction described above

\[
1 / \sqrt{x} \approx y = 2^{-e'} \times y_t,
\]
and start the iteration with \( x_0 = x y^2 \) and \( y_0 = y \). Since \( x_0 = x_m y_t^2 = 1 + x_s / x_t \) in terms of our earlier notation, we conclude that \( |\delta_0| \leq 2^{-b} \) if the table index \( \langle e''|t \rangle \) is \( b \) bits wide. Using the approximation \( \delta_{k+1} \approx 3/4 \times \delta_k^2 \), we obtain \( \delta_3 \approx 5 \times 10^{-16} \) for \( b = 6 \) and \( \delta_3 \approx 6 \times 10^{-18} \) for \( b = 7 \), assuming that \( \delta_0 = 2^{-b} \). Consequently, we will use \( b = 7 \) in the algorithm given below to obtain \( y_3 = 1/\sqrt{x} \) up to numerical errors.

```plaintext
function goldschmidt-1/\sqrt{x} =
\langle s|e'|e''|t|\rangle = x
  \text{--- extract fields}
  x_t = 2^{e'} \times (1.t)[0\ldots0]
  \text{--- do not really compute } x_t, \text{ but } \\
y_t = 1/\sqrt{x_t}
  \text{--- \ldots use } \langle e''|t \rangle \text{ for table lookup}
  y = 2^{-e'} \times y_t
  x = x \times y^2
  \text{--- } x \approx 1 \text{ now}
for k = 1 \ldots 3 do
  r = 3/2 - x/2
  y = y \times r
  x = x \times r^2
od
return y
```

Of course, the loop over \( k = 1 \ldots 3 \) is unrolled in the actual implementation, and the last \( x \times r^2 \) is unused and may be eliminated. Furthermore, the term \( x/2 \) is not explicitly computed each step, but the factors \( 1/2 \) are combined into one factor for the final \( y \).

### Binomial algorithm

To compute \( 1/\sqrt{x} \) for \( x \approx 1 \) we may use the binomial series

\[
(1 + z)^{\alpha} = \sum_{k=0}^{\infty} \binom{\alpha}{k} z^k
\]

with \( \alpha = -1/2 \), which converges absolutely for \( |z| < 1 \). For arbitrary \( x \) we apply the argument reduction steps explained above to find

\[
1/\sqrt{x} = 2^{-e'} \times y_t \times (1 + z_s)^{-1/2}
\]

for small \( z_s = x_s y_t^2 \). We may therefore use a few terms of the binomial series or some other polynomial approximation to compute

\[
y_s = (1 + z_s)^{-1/2} \approx \sum_{k=0} p_k z_s^k,
\]

the coefficients \( p_k \) of the polynomial being determined by minimizing the error in the Chebyshev sense, for instance. Although the formula \( 1/\sqrt{x} = 2^{-e'} \times y_t \times y_s \) is analytically correct, it would not be particularly accurate for floating point arithmetic.

To obtain more accuracy we have to make a few changes. Instead of computing \( y_s = (1 + z_s)^{-1/2} \approx 1 \), we compute \( u_s = y_s - 1 \) directly using

\[
u_s = y_s - 1 = \sum_{k=1} p_k z_s^k,
\]
assuming that the initial coefficient $p_0 = 1$. Furthermore, we need the value $1/\sqrt{\alpha_1}$, which we find by table lookup, with some extra precision. Two simple options for representing a number with extra accuracy are the sum $1/\sqrt{\alpha_1} = y_h + \delta_t$ or the product $1/\sqrt{\alpha_1} = y_h(1 + \epsilon_t)$. We could use either one, but we choose the product representation for technical reasons. The refined formula may then be written

$$\frac{1}{\sqrt{x}} = 2^{-\epsilon_t} \times y_h(1 + \epsilon_t) \times (1 + u_s)$$

where we have neglected the small contribution $\epsilon_t u_s$ in the approximation $(1 + \epsilon_t)(1 + u_s) \approx (1 + \epsilon_t + u_s)$. Note that $y_h(1 + \delta)$ and $y_h + y_h \delta$ are not necessarily the same in floating point arithmetic, and for small $\delta$ and exact $y_h$ the latter expression is often more accurate. Consequently, in our problem it would not be wise to use the apparently equivalent formula $y_h(1 + \epsilon_t + u_s)$.

We may write down now the complete algorithm to compute $1/\sqrt{x}$ using the binomial series for $x \approx 1$. Note that the algorithm does not eliminate the exponential factors in the same place as the derivation, but we still want to use the same terminology. This can be achieved by inserting explicit scaling factors like $2^{2\epsilon_t}$ for associated variables.

```python
def binomial_inv_sqrt(x):
    s, e, m = extract_fields(x)
    x_t = x - 2**e * (1.0|0...0)
    y_t = 1/\sqrt{\alpha_t}
    x_s = x_t * 2**e
    y_s = y_t * x_s
    z_s = x_s * y_s
    u_s = -1/2 * z_s + 3/8 * z_s**2 - ... - 1 + u_s = 1/\sqrt{1 + z_s}
    y = y_t + y_t (\epsilon_t + u_s)
    return y
```

### 2.2.2 Accuracy

In this subsection we analyze the accuracy of Goldschmidt’s algorithm and the binomial algorithm to compute $1/\sqrt{x}$. For reasons of efficiency, both optimized algorithms do not handle arbitrary arguments $x$, but impose some restrictions. The argument $x$ must be a normalized floating point number $x > 0$, and none of the special floating point values like NaN, $\pm \infty$, ... is allowed. This is clearly a disadvantage compared to a standard library approach, but the $1/\sqrt{x}$ computed by the optimized routines for these exceptional $x$ may still be useful. For a given $x = (s|e|m)$, the optimized routines will always interpret this floating point number as

$$x = 2^e \times 1.m,$$

so the routines are actually computing $1/\sqrt{|x|}$. The optimized routines will treat denormalized $x$ and $x = 0$ as very small numbers $x = 2^{-1023} \times (1.+)$, and NaN, $\pm \infty$, ... as very large numbers $2^{1024} \times (1.+).$ This behavior is clearly undesirable for $x = NaN$, but it may be acceptable for the other cases.
We have seen that the algorithms reduce the argument $x$ to the interval $1 < x < 4$ first, and may simply scale the result by the factor $2^{-e}$. Since there is no numerical error involved in computing $2^{-e} \times y$, it suffices to analyze the accuracy of the computed $1/\sqrt{x}$ for $1 < x < 4$. We will compare our routines to a standard $1/sqrt$ routine, which computes $z = \sqrt{x}$ using the math library and $y = 1/z$ in hardware. The absolute error $\epsilon_{abs}$ to the exact $1/\sqrt{x}$ is given both in statistical notation and in graphical form in the table below for Goldschmidt's algorithm and the standard approach. The $x$ values for the analysis are chosen from a uniformly distributed variable in the given interval $1 < x < 4$.

<table>
<thead>
<tr>
<th>Goldschmidt $1/\sqrt{x}$</th>
<th>standard $1/\sqrt{x}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon_{abs} = -2.2 \times 10^{-19} \pm 8.0 \times 10^{-17}$</td>
<td>$\epsilon_{abs} = -7.8 \times 10^{-19} \pm 4.6 \times 10^{-17}$</td>
</tr>
</tbody>
</table>

We see that neither of these routines is accurate to machine precision, which would mean $|\epsilon_{abs}| \leq \epsilon_m/4 \approx 0.55 \times 10^{-16}$. In the standard approach, both $z = \sqrt{x}$ and $y = 1/z$ are computed to machine precision in isolation, but this is evidently not good enough to obtain full accuracy for $1/\sqrt{x}$. Nevertheless, the typical error $\sigma_e$ using the standard approach is only about half the error of Goldschmidt’s algorithm. For many applications a typical error of $\sigma_e \approx 8 \times 10^{-17}$ may still be acceptable. A positive characteristic of Goldschmidt’s algorithm is that, like for the standard approach, $\mu_e$ is small, which means that there is no overall bias. Furthermore, inspection of the graphics does not show any obvious unwanted structure in the error. Of course, it is still possible that Goldschmidt’s algorithm is biased for arguments $x$ within small subintervals, despite the fact that $\mu_e$ is small for the interval $1 < x < 4$.

We will analyze the error for the binomial algorithm to compute $1/\sqrt{x}$ now. The absolute error $\epsilon_{abs}$ to the exact $1/\sqrt{x}$ is given both in statistical notation and in graphical form in the table below for the accurate binomial $1/\sqrt{x}$ and the standard approach mentioned above. Again, the $x$ values for the analysis are chosen from a uniformly distributed variable in the given interval $1 < x < 4$.

<table>
<thead>
<tr>
<th>Binomial $1/\sqrt{x}$</th>
<th>standard $1/\sqrt{x}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon_{abs} = -1.8 \times 10^{-18} \pm 3.2 \times 10^{-17}$</td>
<td>$\epsilon_{abs} = -7.8 \times 10^{-19} \pm 4.6 \times 10^{-17}$</td>
</tr>
</tbody>
</table>
We see that the $1/\sqrt{x}$ computed by the binomial algorithm are at least close to machine precision, since $|e_{abs}| \leq e_m / 4 \simeq 0.55 \times 10^{-16}$ most of the time. The overall error $\sigma_e$ using the standard approach is about $1.5$ times larger compared to the binomial algorithm. Of course, few applications will ever care about the difference between $\sigma_e \simeq 3.2 \times 10^{-17}$ and $\sigma_e \simeq 4.6 \times 10^{-17}$.

### 2.2.3 Optimization

We have seen two sequential algorithms to compute $1/\sqrt{x}$ and analyzed the accuracy until now. In this subsection we explain how to implement Goldschmidt’s algorithm to compute $1/\sqrt{x}$ efficiently by applying simple optimization techniques. We will not describe here the analogous optimization steps for the binomial algorithm, so in this subsection computing $1/\sqrt{x}$ will always mean computing $1/\sqrt{x}$ using Goldschmidt’s algorithm.

Of course, we assume that we want to compute $1/\sqrt{x}$ not for a single value $x$ but for a vector $(x_k)$ of input arguments. One of the most useful initial steps to analyze the efficiency of an algorithm is the construction of the dependence graph. Since the computation of $1/\sqrt{x_k}$ may occur independently for different $x_k$, the dependence graph shown below for the computation of a single $1/\sqrt{x}$ contains the necessary information. For convenience, the graph is shown from left to right.

A straightforward instruction count shows that there are $10 \mathbin{\oplus}, 3 \mathbin{\ominus}, 4 \mathbin{\otimes}, 2 \mathbin{\odot}, 3 \mathbin{\oslash}$ and $2 \mathbin{\oslash}$ for each $1/\sqrt{x}$. Often we do not need this in detail, but it is good enough to count the total number of integer instructions, in our case we would have $11 \mathbin{\oplus\ominus\otimes\odot\oslash\oslash}$. Ignoring other restrictions for the moment, the efficiency is therefore limited by the $\mathbin{\oplus}$ instructions, and the computation of a single $1/\sqrt{x}$ will take at least $10$ cycles. To achieve this optimum of $10$ cycles we will ultimately have to schedule instructions in such a way that one $\mathbin{\oplus}$ is executed in each cycle. The issue map of the sequential code given below shows that we cannot achieve this goal by instruction scheduling within a single iteration.

Since the result delay of both floating point addition and multiplication is $4$ cycles, it is often appropriate to unroll the loop $4$ times and compute $4$ independent $1/\sqrt{x}$ more or less in parallel. One important consequence of loop unrolling is that it is easier now to achieve the optimal rate of one $\mathbin{\oplus}$ per cycle, since each of the $4$ instances has to issue one $\mathbin{\oplus}$ every fourth cycle. Given the above dependence graph, this can be done by ordering the multiplications as follows.
Let us make the simplifying assumption that \( n \), the number of \( 1 / \sqrt{x} \) to be computed, is a multiple of 4. Then the loop may be unrolled 4 times and we may apply instruction scheduling to the resulting code. This process is shown in the issue maps below.

We see from the issue map after loop unrolling and scheduling that each iteration consists of a phase containing primarily integer operations, and a subsequent phase with floating point operations. Such an issue pattern is well suited for software pipelining, as can be seen in the issue maps below. To see the transformation involved in the threefold software pipelining more clearly, we have colored the 3 phases differently. The pre-loop and post-loop parts introduced by the software pipelining step are not of particular interest, so we only indicate their presence in the issue map.

Note that the first phase contains only one \( \cdot \) from each instance, so one might think that twofold software pipelining would be sufficient. In fact, it would be quite easy to use only twofold software pipelining, but if one considers eventual load latencies it is usually advantageous to issue loads early.
While counting cycles based on the instruction sequence may be misleading because we ignore memory latencies, it still determines an upper limit for the possible improvement thanks to the code transformations. Initially, we had $54$ cycles for $n$ iterations of the loop or $54n$ overall. After loop unrolling we had $67/4 \times 17n$ cycles, and after software pipelining we have about $40/4 \times 10n$ cycles plus some overhead. So these simple code transformations alone promise to make the algorithm run faster by about a factor of $5$.

### 2.2.4 Efficiency

We have seen above that Goldschmidt's algorithm requires $10$ cycles per $1/\sqrt{x}$ for the optimized implementation, if we ignore any overhead. With a clock rate of $500$ MHz, these $10$ cycles correspond to $20$ ns as a lower limit for computing one $1/\sqrt{x}$. Of course, the true execution time $t$ will be higher than $20$ ns per $1/\sqrt{x}$ for several reasons. The branch will be mispredicted at least once, and even correctly predicted taken branches involve some overhead. Furthermore, loads from the argument vector $(x_k)$ or from the lookup table may miss the first-level cache. The cost of function call and return, including the code to save and restore register values, may be substantial. Finally, with software pipelining we achieve high performance in the loop itself, but non-optimal pre-loop and post-loop blocks cannot be avoided.

We will use the idealized model for the execution time $t = t_0 + c_1 + c_2/n$ with $t_0 = 20$ ns to analyze the efficiency depending on the vector length $n$. The execution times $t$ for Goldschmidt's algorithm are shown below for a couple of vector lengths $n$. The times $t$ are given in ns per $1/\sqrt{x}$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$t[\text{ns}]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>50.0</td>
</tr>
<tr>
<td>8</td>
<td>35.4</td>
</tr>
<tr>
<td>16</td>
<td>28.1</td>
</tr>
<tr>
<td>32</td>
<td>24.5</td>
</tr>
<tr>
<td>64</td>
<td>23.8</td>
</tr>
<tr>
<td>128</td>
<td>21.7</td>
</tr>
<tr>
<td>256</td>
<td>21.6</td>
</tr>
</tbody>
</table>

Fitting this data to our simple model, we obtain $c_1 \approx 1.2$ ns or about $0.6$ cycles overhead per $1/\sqrt{x}$, and $c_2 \approx 115$ ns overhead per vector. In relation to the idealized execution time of $20$ ns the overall overhead drops to about $6\%$ for $n \to \infty$. We will see later that the cost for a single $1/\sqrt{x}$ using the standard library is $> 150$ ns. If we want to compute $1/\sqrt{x}$ for a vector length $n = 2$, the standard approach will require $> 300$ ns whereas Goldschmidt's algorithm takes only $200$ ns even for $n = 4$.

To perform an analogous execution time analysis for the binomial algorithm, we need to know the idealized execution time $t_0$ first. The instruction count for the binomial algorithm results in $11 \odot, 9 \odot$ and $17 \odot\odot\odot$ with a minimum of $11$ cycles required. The optimized implementation of the binomial $1/\sqrt{x}$ achieves this lower limit in its loop, so we have $t_0 = 22$ ns. The execution times $t$ for the binomial algorithm are shown below for varying vector length $n$. The times $t$ are given in ns per $1/\sqrt{x}$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$t[\text{ns}]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>50.0</td>
</tr>
<tr>
<td>8</td>
<td>35.4</td>
</tr>
<tr>
<td>16</td>
<td>28.1</td>
</tr>
<tr>
<td>32</td>
<td>24.5</td>
</tr>
<tr>
<td>64</td>
<td>23.8</td>
</tr>
<tr>
<td>128</td>
<td>21.7</td>
</tr>
<tr>
<td>256</td>
<td>21.6</td>
</tr>
</tbody>
</table>
Fitting this data to our simple model, we obtain $c_1 = 2.4\,\text{ns}$ or about 1.2 cycles overhead per $1/\sqrt{x}$, and $c_2 = 318\,\text{ns}$ overhead per vector. Compared to the idealized execution time $t_0 = 22\,\text{ns}$ the overall overhead drops to about 11% for $n \to \infty$. Note that both overheads $c_1$ and $c_2$ for the binomial algorithm are much higher than for Goldschmidt’s algorithm. The main reasons for this significant change are the larger lookup table, the higher degree of software pipelining, the number of loads within the loop and the number of floating point registers required.

We will look now at the execution times of Goldschmidt’s algorithm on other processors than the Alpha 21164. Although the implementation is optimized for this particular processor, there are still a couple of reasons to try out other processors. First, we see that the manual optimization targeted at a particular processor can make a big difference. Second, some C compilers can be seen failing to compile the sequential version of Goldschmidt’s algorithm efficiently. Third, for some processors the Goldschmidt $1/\sqrt{x}$ implementation may still turn out to be faster than an approach based on the standard library or even hardware. An exceptional feature of the manually optimized code is that further optimizations by the compiler may actually decrease its efficiency. We will draw a few conclusions after the following table which displays execution times $t$ per $1/\sqrt{x}$ for a vector length $n = 100$. The method denoted as standard computes $y = 1/\sqrt{x}$ by using $z = \sqrt{x}$ and then $y = 1/z$. We do not insist on particularly accurate results, so the compiler or linker is free to use a math library routine $\text{sqrt}$ which values efficiency more than accuracy.

<table>
<thead>
<tr>
<th>processor</th>
<th>$t$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164</td>
<td>22</td>
</tr>
<tr>
<td>[500 MHz]</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>130</td>
</tr>
<tr>
<td></td>
<td>163</td>
</tr>
<tr>
<td>UltraSPARC-II</td>
<td>60</td>
</tr>
<tr>
<td>[336 MHz]</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>144</td>
</tr>
<tr>
<td></td>
<td>143</td>
</tr>
<tr>
<td>MIPS R10000</td>
<td>62</td>
</tr>
<tr>
<td>[195 MHz]</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>114</td>
</tr>
<tr>
<td></td>
<td>182</td>
</tr>
</tbody>
</table>

---

The table shows the execution times $t$ per $1/\sqrt{x}$ for a vector length $n = 100$. The methods are:

- Standard $1/\sqrt{x}$ (gcc -fast)
- Optimized Goldschmidt $1/\sqrt{x}$ (gcc -01)
- Optimized Goldschmidt $1/\sqrt{x}$ (cc -fast)
- Sequential Goldschmidt $1/\sqrt{x}$ (cc -fast)

These times reflect the performance of Goldschmidt’s algorithm on different processors, illustrating the variations in efficiency due to compiler optimizations and hardware differences.
We see that the optimized Goldschmidt $1/\sqrt{x}$ on the Alpha is fastest, followed by the UltraSPARC-II, which is slower by about a factor 2.5. The difference between using gcc -01 and cc -fast is smaller than we expected, and is the most significant for the relatively old UltraSPARC. The efficiency of the sequential version depends on the quality of the compiler used, being about 1.8 times slower for the MIPS and a factor 5.9 slower on the Alpha compared to the optimized routine. Most important, the optimized $1/\sqrt{x}$ is much faster than the standard library approach for all newer processors in the table above.

### 2.3 Computing $1/x$

In this subsection we describe how to compute $1/x$ efficiently. Fortunately, we may reuse some notation from the $1/\sqrt{x}$ case, so the following explanation can be kept concise. Again, we will end up with two algorithms to compute $1/x$, one emphasizing efficiency at the cost of accuracy, the other being accurate close to machine precision. Interestingly, both algorithms are based on Goldschmidt’s idea. This division algorithm is apparently named after [11], and a short description of an early implementation in hardware is given in [2]. A newer description of Goldschmidt’s idea and an introduction to computer arithmetic in general can be found in [10]. We will explain the algorithms first and plunge into questions of accuracy and efficiency later.

#### 2.3.1 Algorithm

For a floating point number $x = (s|e|n)$ corresponding to the value $x = (-1)^s \times 2^e \times (1.m)$, we have

$$1/x = (-1)^s \times 2^{-e} \times 1/x_m \quad \text{with} \quad x_m = (1.m).$$

Since the factor $(-1)^s \times 2^{-e}$ poses no problem, we have reduced the problem to the interval $1 \leq x_m < 2$. To further reduce the argument $x_m$ we find an approximation

$$x_t = (1.t|0\ldots0) \approx x_m = (1.m),$$

where $t$ contains the first few bits of $m$, so that $y_t = 1/x_t$ may be determined by table lookup. We may then write

$$1/x = (-1)^s \times 2^{-e} \times 1/x_t \times 1/(x_m/x_t)$$

$$= (-1)^s \times 2^{-e} \times y_t \times (x_m y_t)^{-1},$$
and since \( x_t \approx x_m \) we are essentially left with the problem of computing \( 1/u \) for \( u = x_m y_t \approx 1 \).

Goldschmidt's algorithm to compute \( 1/x \) is very similar to the algorithm we have seen earlier to compute \( 1/\sqrt{x} \). The idea is to find a sequence

\[
x = x_0 = x_1 = \ldots = x_k = \ldots
\]

such that \( x_k \to 1 \) and therefore \( y_k \to 1/x \). In each step \( k \to k+1 \) we determine \( r_k \approx 1/x_k \) and let \( x_{k+1} = x_k r_k \) and \( y_{k+1} = y_k r_k \). Assuming that \( x_k = 1 + \delta_k \) for some small \( \delta_k \), we may use the approximation \( r_k = 1 - \delta_k = 2 - x_k \). Then the step \( x_k \to x_{k+1} \) may be written

\[
x_{k+1} = x_k r_k = (1 + \delta_k)(1 - \delta_k) = 1 - \delta_k^2 = 1 + \delta_{k+1},
\]

so \( (x_k) \) converges quadratically to 1. To obtain a \( x_0 \approx 1 \) for arbitrary \( x \), we may use the approximation from the argument reduction described above

\[
1/x \approx y = (-1)^s \times 2^{-e} \times y_t,
\]

and start the iteration with \( x_0 = x y \) and \( y_0 = y \). If the table index \( t \) is \( b = 6 \) bits wide, we may conclude that \( |\delta_0| \leq 2^{-7} \). Using the relation \( \delta_{k+1} = -\delta_k^2 \) and assuming a maximal \( \delta_0 = 2^{-7} \), we obtain \( |\delta_3| = 2^{-56} = 1.4 \times 10^{-17} \), which we consider small enough for the efficient algorithm. Consequently, we will use \( b = 6 \) in the algorithm given below to obtain \( y_3 = 1/x \) up to numerical errors

```c
function goldschmidt-1/x =
    \langle s|e|t|+\rangle = x    — extract fields
    x_t = \langle 1.t|1|0\ldots 0 \rangle    — do not really compute \( x_t \), but...
    y_t = 1/x_t    —... use \( t \) for table lookup
    y = (-1)^s \times 2^{-e} \times y_t
    x = x \times y    — \( x \approx 1 \) now
    for k = 1\ldots 3 do
        r = 2 - x
        y = y \times r
        x = x \times r
    od
    return y
```

Of course, the loop over \( k = 1\ldots 3 \) is unrolled in the actual implementation, and the last \( x \times r \) is unused and may be eliminated. An instruction count for the optimized implementation of Goldschmidt’s algorithm to compute \( 1/x \) gives \( 6 \oplus, 3 \oplus \) and \( 11 \ominus \ominus \ominus \ominus \) with a minimum of 6 cycles required.

We do not expect Goldschmidt’s algorithm to be particularly accurate for a couple of reasons. First and most important, each multiplication \( y \times r \) adds some numerical error, both from the inaccuracy of \( r \) and the multiplication itself. Second, the approximation \( y_t \) is only accurate to machine precision. Third, for \( b = 6 \) we may still have \( \delta \approx 10^{-17} \) after 3 iterations, which is not small enough for rounding close to machine precision. The last problem is easy to solve
Computing $1/x$

by letting $b = 7$, therefore doubling the size of the lookup table. The other problems may be
cured by using some extra precision. Of course, if hardware supports floating point arithmetic
with some extra precision, there is no problem, but we may solve the problem in software as
well.

The key ingredients for getting extra precision are the following. First, we represent $1/x_i$ to
more precision using $1/x_i = y_i(1 + \epsilon_i)$ for floating point numbers $y_i$ and small $\epsilon_i$. Second,
instead of using $x_k$ and $y_k$ directly in the algorithm, we use smaller corrections $\delta_k = x_k - 1$
and $z_k = y_k - y_t$. We may then translate the expressions $r_k = 2 - x_k$ as well as $x_{k+1} = x_k r_k$
and $y_{k+1} = y_k r_k$ in a fairly mechanical way to obtain

\[
\begin{align*}
\delta_{k+1} &= x_{k+1} - 1 = x_k r_k - 1 = (1 + \delta_k)(1 - \delta_k) - 1 = -\delta_k^2 \\
z_{k+1} &= y_{k+1} - y_t = y_k r_k - y_t = (z_k + y_t)(1 - \delta_k) - y_t = z_k - \delta_k(z_k + y_t)
\end{align*}
\]

We will call the resulting algorithm accurate $1/x$ for lack of an appropriate name, since it still
implements only Goldschmidt’s idea. As mentioned above, the index $t$ for the table lookup is
$b = 7$ bits wide for the accurate $1/x$.

```plaintext
function accurate-1/x =
  \langle s | e | f | t | \rangle = x          --- extract fields
  x_t = \langle t | 1 | 0 | \ldots | 0 \rangle
  x_s = (-1)^s \times 2^{-e} \times x - x_t
  y_t(1 + \epsilon_t) = 1/x_t          --- use index $t$ for table lookup
  \delta = x_s \times y_t
  z = 0
  for $k = 1 \ldots 3$ do
    z = z - \delta(z + y_t)
    \delta = -\delta^2
  od
  y = y_t + (y_t \epsilon_t + z)
  y = (-1)^s \times 2^{-e} \times y
return y
```

Of course, the loop over $k = 1 \ldots 3$ is unrolled in the actual implementations, and a few
simplifications are possible. The last $\delta = -\delta^2$ is unused, and the sign of $-\delta^2$ may be taken
into account in the addition later, thereby saving one multiplication. Furthermore, the first $z =
z - \delta(z + y_t)$ simplifies to $z = -\delta y_t$, and again it suffices to compute $\delta y_t$. Another difference
of the formal description above to the actual implementation is that the factor $(-1)^s \times 2^{-e}$
is inserted earlier in the implementation for technical reasons. An instruction count for the
optimized implementation of the accurate $1/x$ gives 7 $\times$, 7 $\div$ and 16 $\times\times$ with a minimum
of 8.5 cycles, if we take into account the loop code as well. Our implementation of the accurate
$1/x$ wastes another 0.5 cycles, so it requires at least 9 cycles per $1/x$.

2.3.2 Accuracy

For efficiency reasons, the optimized algorithms to compute $1/x$ impose some restrictions on
the argument $x$. For a given $x = \langle s | e | n \rangle$, they will always interpret this floating point number
as
\[ x = (-1)^s \times 2^e \times (1.m), \]
so the routines will fail for denormalized \( x \) and also for \( |x| > 2^{1022} \) due to underflow. The
failure for very large \( x \) or for \( x = \text{NaN}, \pm \infty, \ldots \) may lead to completely wrong results or even
floating point exceptions. The cases \( x = 0 \) or denormalized \( x \) will look like very small numbers
to the optimized routines, which may be acceptable in some applications.
We have seen that the algorithms reduce the problem of computing \( 1/x \) to the interval \( 1 \leq x < 2 \) first, and may compute the result for arbitrary cases without further error. Therefore it
is sufficient to analyze accuracy for the interval \( 1 \leq x < 2 \). This absolute error \( \epsilon_{\text{abs}} \) is given in
the table below both in statistical notation and in graphical form for the optimized algorithms
to compute \( 1/x \). The \( x \) values for the analysis are chosen from a uniformly distributed variable
in the interval \( 1 \leq x < 2 \).

<table>
<thead>
<tr>
<th>Goldschmidt ( 1/x )</th>
<th>accurate ( 1/x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \epsilon_{\text{abs}} = 3.1 \times 10^{-18} \pm 9.3 \times 10^{-17} )</td>
<td>( \epsilon_{\text{abs}} = -5.5 \times 10^{-10} \pm 3.2 \times 10^{-17} )</td>
</tr>
<tr>
<td>( \epsilon_{\text{abs}} )</td>
<td>( \epsilon_{\text{abs}} )</td>
</tr>
<tr>
<td>( -2 \times 10^{-16} )</td>
<td>( -2 \times 10^{-16} )</td>
</tr>
<tr>
<td>( x = 1 )</td>
<td>( x = 1 )</td>
</tr>
<tr>
<td>( x = 2 )</td>
<td>( x = 2 )</td>
</tr>
</tbody>
</table>

We see that the \( 1/x \) computed by the accurate algorithm are close to machine precision, since
\( |\epsilon_{\text{abs}}| \leq \epsilon_m / 4 = 0.55 \times 10^{-16} \) most of the time. The overall error \( \sigma_e \) using Goldschmidt’s
algorithm is about 3 times larger than \( \sigma_e \) for the accurate \( 1/x \). Note that although the accurate \( 1/x \) is always close to machine precision, it does not round correctly as required by the
IEEE 754 standard for floating point division, so hardware \( 1/x \) is still more accurate by a tiny
bit.

2.3.3 Optimization
We have already mentioned the instruction counts for the two algorithms, so we will only show
the dependence graph and the issue maps for Goldschmidt’s algorithm to compute \( 1/x \) below.
The dependence graph for \( 1/x \) looks similar to the one for \( 1/\sqrt{x} \). The main differences are
that we do not have to shift the exponent \( e \) and that we compute \( x \times r \) instead of \( x \times r^2 \) in each step.

We obtain the instruction count given earlier with 6 \( \text{GXX} \), 3 \( \text{LXX} \) and 11 \( \text{LXX} \). The issue maps
shown below sketch the optimization steps from the sequential to the optimized implementation.
We may quickly check the correctness of the instruction count given earlier by counting instructions in the issue map. In the fourfold unrolled code, we count 24 ©, 12 © and 48 ©©© per loop, including the 4 ©©©©©© from the loop code.

2.3.4 Efficiency

We have seen that Goldschmidt’s algorithm to compute $1/x$ can be implemented so that it requires 6 cycles per $1/x$, if we ignore any overhead. Although we have not shown it here, our implementation of the accurate algorithm requires at least 9 cycles. To analyze the efficiency of these algorithms for varying vector lengths $n$, we use again the simple model for the execution time $t = t_0 + c_1 + c_2/n$ with $t_0 = 12 \, ns$ for Goldschmidt’s algorithm and $t_0 = 18 \, ns$ for the accurate $1/x$. The measurement of the execution time $t$ for Goldschmidt’s algorithm is given below for varying vector lengths $n$. As usual, the execution time $t$ is given in $ns$ per $1/x$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$t[ns]$</th>
<th>$c_1 = 12 , ns$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>62.5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>41.6</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>28.1</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>20.3</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>17.5</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>14.7</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>13.9</td>
<td></td>
</tr>
</tbody>
</table>

Fitting this data to our simple model, we obtain $c_1 \doteq 1.5 \, ns$ or about 0.8 cycles overhead per $1/x$, and $c_2 \doteq 212 \, ns$ overhead per vector. In relation to the idealized execution time of $12 \, ns$ the overall overhead drops to about 12% for $n \to \infty$.

The analogous measurements and analysis for the accurate algorithm to compute $1/x$ are given below.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$t[ns]$</th>
<th>$c_1 = 18 , ns$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>62.5</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>45.9</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>33.3</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>26.0</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>24.5</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>22.9</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>21.3</td>
<td></td>
</tr>
</tbody>
</table>
Fitting this data to our simple model, we obtain $c_1 = 3.2 \text{ ns}$ or about 1.6 cycles overhead per $1/x$, and $c_2 = 176 \text{ ns}$ overhead per vector. In relation to the idealized execution time of 18 ns the overall overhead drops to about 18% for $n \to \infty$.

Finally, we try Goldschmidt's algorithm for a vector length $n = 100$ on other processors and for different compiler flags to see how much of a difference this makes. The method denoted as $\text{hardware}$ computes $1/x$ using the hardware floating point divide, obviously.

<table>
<thead>
<tr>
<th>processor</th>
<th>$t [\text{ ns}]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164</td>
<td>15</td>
</tr>
<tr>
<td>[500 MHz]</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>70</td>
</tr>
<tr>
<td>UltraSPARC-II</td>
<td>109</td>
</tr>
<tr>
<td>[336 MHz]</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>76</td>
</tr>
<tr>
<td>MIPS R10000</td>
<td>48</td>
</tr>
<tr>
<td>[195 MHz]</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>88</td>
</tr>
<tr>
<td></td>
<td>111</td>
</tr>
</tbody>
</table>

As expected, the algorithm runs much faster on the Alpha than on other processors. The most important conclusion we may draw is that computing $1/x$ in software may be worthwhile for other processors as well, provided that we have to compute several $1/x$ at a time and we do not care about very large or even exceptional values $x$. Depending on the accuracy required, we may then choose either the fast but not particularly accurate Goldschmidt $1/x$ or the accurate $1/x$.

### 2.4 Computing $e^x$

The exponential function $e^x$ is usually computed by calling a library routine $\exp$, which may be assumed to compute $e^x$ efficiently and accurately. Furthermore, the library routine will handle exceptional cases like underflow, overflow or input arguments like $-\infty$ gracefully. In our particular problem we are interested in both $e^x$ and $e^{-x}$, so we will explain in this subsection how to compute pairs $e^{\pm x} = (e^x, e^{-x})$ efficiently for several values of $x$.

First, we will describe the algorithm and analyze its accuracy. We will see that the loss in accuracy is negligible for $e^{\pm x}$ compared to the standard library routine $\exp$. Then we explain the optimization steps applied, and finally we analyze the efficiency of our implementation.
2.4.1 Algorithm

In this subsection we explain the algorithm to compute \( e^x \) which can then be easily extended to compute \( e^{-x} \) at the same time. To understand the algorithm and particularly its subtleties, you should feel comfortable with the floating point format. A short introduction to the IEEE 754 floating point standard is given in Appendix A.

For small \(|x|\) the exponential function \( e^x \) can be efficiently computed by using a couple of terms of the Taylor series

\[
e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!},
\]

or more generally by any polynomial \( p(x) = \sum_k p_k x^k \) which approximates \( e^x \) within a given interval \(-\xi < x < \xi\). Note that an approximation for the function \( e^{-x} \) is then given by \( p(x) = \sum_k (-1)^k p_k x^k \), so we may use the terms \( p_k x^k \) for both \( e^x \) and \( e^{-x} \). For larger \(|x|\) the Taylor series still converges, but the number of terms required would make this approach very inefficient, and the result would not be particularly accurate.

Internally, floating point numbers are stored as tuples \( (s|e|m) \) corresponding to the value \((-1)^s \times 2^e \times (1.m)\), if we ignore the bias in the exponent \( e \) for now. Therefore, it is natural to write \( e^x = 2^y/\ln 2 = 2^y \) and split \( y \) into its integral part \( y_e = \lfloor y \rfloor \) and fractional part \( y_f \) such that \( y = y_e + y_f \). Using the addition theorem for exponential functions \( e^{a+b} = e^a e^b \) we obtain

\[
e^x = 2^{y_e} \times 2^{y_f},
\]

where \( y_e \) is an integer and \( 0 \leq y_f < 1 \). To compute \( 2^{y_e} \) in the floating point format, we simply have to fill the exponent field with \( y_e \), so we are left with the problem of computing \( 2^{y_f} = e^{y_f/\ln 2} \). Since \( 0 \leq y_f \ln 2 < \ln 2 \), we could apply the Taylor series now, but it is more efficient to split \( y_f \) as well. Let \( y_f = (0.f) = (0.t|s) \) be a splitting with \( t \) being \( b \) bits wide, and define \( y_t = (0.t) \) and \( y_s = (0.0|s) \) such that \( y_f = y_t + y_s \). Obviously, we have

\[
2^{y_f} = 2^{y_t} \times 2^{y_s},
\]

where \( y_t = (0.t) \) with \( 0 \leq t < 2^b \) and \( 0 \leq y_t \ln 2 < 1 \). The value \( 2^{y_t} \) may be obtained by table lookup with the index \( t \), and a polynomial approximation of low degree may be used for \( 2^{y_s} = e^{y_s/\ln 2} \). We may decrease the polynomial degree required by increasing \( b \) and therefore increasing the size of the table.

To summarize, the exponential function \( e^x \) may be computed by splitting \( y = x/\ln 2 \) into \( y = y_e + y_t + y_s \) and using the identity

\[
e^x = 2^y = 2^{y_e} \times 2^{y_t} \times 2^{y_s},
\]

where \( y_e \) is integral, \( y_t \) is described by a few bits, and \( y_s \) is small. We have just seen one particular way to split \( y \), now let us describe a more clever method. One ingredient of this method is the conversion of \( y \) into a fixed point representation within the mantissa of a floating point number. Let us assume for now that \( 0 \leq y \ll 2^{52-b} \) such that \( 2^{52-b} \leq \bar{y} = 2^{52-b} + y < 2^{53-b} \) after rounding \( \bar{y} \) to a floating point representation. In other words, we know that the exponent of \( \bar{y} \) is \( e = 52-b \) and that the mantissa \( \bar{m} \) contains the fractional part of \( \bar{y} \) in its last \( b \) bits. A part of the splitting of \( y \) may then be obtained via \( \langle 1.\bar{m} \rangle = \langle \cdots e|t \rangle \) with \( y_e = \langle c \rangle \) and...
The restriction \( y \ll 2^{52-b} \) is no problem for reasonable \( b \), since \( e^x \) will overflow for \( y \geq 2^{10} \) anyway, but we must be able to handle the general case \(-2^{10} < y < 2^{10}\). We compute \( \bar{y} = 3 \times 2^{51-b} + y \) and find again that \( 2^{52-b} \leq \bar{y} < 2^{53-b} \) after rounding. For \( y \geq 0 \) the mantissa of \( \bar{y} \) is given by \((\bar{m}_{\bar{y}}) = (1.\bar{m}_{\bar{y}}) = (1.10...0|e|t)\) and for \( y < 0 \) we have \((\bar{m}_{\bar{y}}) = (1.\overline{01}...1|e|t)\), unless \( y \approx 0 \) and \( \bar{y} = \bar{y} + y \) numerically. We see that in both cases \( \bar{m} = (\cdots|e|t) \), where the exponent \( e \) is given as a signed integer in two-complement representation, and the table index \( t \) occupies the \( b \) lowest bits of the mantissa.

Once we have determined \((e|t)\) and therefore \( y_h = y_e + y_t \), it would be possible to find \( y_s = y - y_h \) and then compute \( 2^{y_s} \). Analytically, this would be a correct solution, but numerically the value \( y = x/\ln 2 \) is not completely accurate because of the multiplication with an approximation to \( 1/\ln 2 \) and rounding thereafter. It is preferable to compute \( x_s = x - y_h \ln 2 \) accurately and use \( 2^{y_s} = e^{x_s} \). Unless \( x \) is small in absolute value we have \( x \approx y_h \ln 2 \), so we have to watch out for cancellation when computing \( x_s \). Both the product \( y_h \ln 2 \) and the difference \( x - y_h \ln 2 \) may be computed more accurately than usual using the following ideas.

The less significant bits of the mantissa of \( y_h \) are all zero, since we have \((y_{\bar{h}}) = (\bar{m}_{\bar{y}}|0...0)\). This is important because a product \((u|0_u) \times (v|0_v)\) is computed exactly in floating point arithmetic if the analytic result \((uv|0_{uv})\) contains enough zero bits in its least significant part, which is certainly the case if \( (0_u|0_v) \) is at least 54 bits wide. To exploit this property, we represent \( \lambda = \ln 2 \) as a sum of two values \( \lambda = \lambda_h = \lambda - x \), where \( \lambda_h \approx \lambda \) and the mantissa of \( \lambda_h \) looks like \((\cdots|0)\), such that \( y_h \times \lambda_h \) can be computed exactly in floating point arithmetic. Given this splitting \( \ln 2 = \lambda = \lambda_h + \lambda_s \), we may write

\[
x_s = x - y_h \ln 2 = x - y_h (\lambda_h + \lambda_s) = (x - y_h \lambda_h) - y_h \lambda_s.
\]

While these expressions are all equivalent analytically, the last expression is preferred numerically, since \( x \) and \( y_h \lambda_h \) are exact, and since \( x \approx y_h \lambda_h \) the difference \( x - y_h \lambda_h \) is computed exactly as well in floating point arithmetic, so no additional error is introduced up to this point even if cancellation occurs for \( x \neq 0 \). In contrast to this, the term \( y_h \lambda_s \) is not exact, but the error made can be considered negligible since \( \lambda_s \ll \lambda_h \).

There is one last subtlety we need to compute \( e^{x_h} \) accurately, so hold on. Up to now, we may compute \( 2^{y_h} \), \( 2^{y_s} \) and \( e^{x_r} \) accurately, but we really want to compute

\[
e^{x} = 2^{y_h} \times 2^{y_s} \times e^{x_s}.
\]

The factor \( 2^{y_h} \) is no problem since this multiplication can be done by adding \( y_h \) to the exponent field of the floating point representation. We are left with computing \( 2^{y_s} \times e^{x_r} \) accurately, which cannot be done by a simple floating point multiplication, since both \( 2^{y_s} \) and \( e^{x_r} \) are infected by rounding errors and the multiplication itself will add some additional error. Therefore both the factors and the multiplication itself require some extra accuracy. Since \( 2^{y_s} \) is obtained by table lookup, we may store it more accurately by using \( 2^{y_s} = v_t + \delta_t \) or \( 2^{y_s} = u_t (1 + \epsilon_t) \) for small \( \delta_t \) and \( \epsilon_t \), our choice being the latter for technical reasons. We obtain \( e^{x_r} \) with some extra precision by computing \( u_s = e^{x_r} - 1 \) accurately and using the representation \( e^{x_r} = 1 + u_s \), which works because \( u_s \) is small for small \( x_r \). Then we have

\[
2^{y_s} \times e^{x_r} \approx u_t (1 + \epsilon_t) (1 + u_s) = u_t (1 + \epsilon_t + u_s + \epsilon_t u_s)
\]

\[
\approx u_t (1 + \epsilon_t + u_s) = u_t (\epsilon_t + u_s),
\]
where we have ignored the small contribution $\epsilon f u_s$ to obtain the final expression. The last expression is preferred numerically, since $u_t$ is exact and $u_t(\epsilon_1 + u_s) \ll u_t$, so a small error in this second term does not much harm.

We are finally ready to write down the algorithm for computing $e^x$ efficiently and close to machine precision. This algorithm makes some assumptions about the input argument $x$ which we will specify later, and it assumes further that the floating point arithmetic is IEEE 754 compliant. For simplicity we will assume that the number of bits $b = 5$ for the table lookup.

```plaintext
function exp :-
y = x \times 1/\ln2
\hat{y} = y + 3 \times 2^{46}
y_h = y - 3 \times 2^{46}
x_s = (x - y_h)^{\lambda_h} - y_h \lambda_s
epsilon \equiv (\hat{y})_{16.5}
index t for table lookup
u_t(1 + \epsilon_t) = 2^{(0.t)}
table lookup
u_h = u_t \times 2^e
add e to exponent field
u_s = x_s^2/2 + x_s^3/6 + \cdots
exponent e
z = u_h + u_h(\epsilon_1 + u_s)
return z
```

It is important to note that the computation of $y_h$ is only possible as shown here if there is no extra precision in floating point registers. In other cases it may be necessary to write $y$ to memory to reduce the mantissa to 53 bits, unless the hardware supports some flag to always round to 53 bits despite the available extra bits.

### 2.4.2 Accuracy

In this subsection we analyze the accuracy of our optimized routine, basically by comparing it to a reference implementation from the math library. While our primary goal is to compute $e^\pm x$ as quickly as possible, doing this with high accuracy is important as well. For efficiency reasons, the optimized routine to compute $e^\pm x$ assumes that the argument $x$ satisfies the following conditions

1. The argument $x$ may be a denormalized floating point number, but none of NaN, ±\infty, …
2. Neither $x$ nor $-x$ do result in the computed $e^\pm x$ to overflow or being denormalized. This condition is certainly fulfilled if $|x| + \epsilon < 1022 \times \ln 2 \approx 708.4$ for a small margin $\epsilon$, which is necessary to absorb eventual errors in the computation of $e^\pm x$.

If one of these conditions is violated, the result may be completely wrong or may result in a floating point exception.

We have seen that the algorithm reduces the argument $x$ to the interval $0 \leq \hat{x} < \ln 2$ first, and computes $e^x = 2^x \times e^{\hat{x}}$ with integral exponent $e = \lfloor x / \ln 2 \rfloor$. Since the argument reduction $x \rightarrow \hat{x}$ is done accurately, it suffices to analyze the computed $e^{\hat{x}}$ for $0 \leq x < \ln 2$. Naturally, we compare our routine to an `exp` routine in the standard C math library, which we plan to replace by the optimized $e^\pm x$. The absolute error $\epsilon_{abs}$ is given both in statistical notation and
in graphical form in the table below. The \( x \) values for the analysis are chosen from a uniformly distributed variable in the given interval \( 0 < x < \ln 2 \).

<table>
<thead>
<tr>
<th>( x )</th>
<th>( c^{0.5} )</th>
<th>( c^{1.0} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( 1.000000000000000 )</td>
<td>( 1.000000000000000 )</td>
</tr>
<tr>
<td>( \ln 2 )</td>
<td>( 1.648721270700128 )</td>
<td>( 1.648721270700128 )</td>
</tr>
</tbody>
</table>

We see that the chosen \( \exp \) library routine computes \( e^x \) very accurately, since the absolute error \( |e_{\text{abs}}| \leq e_{m}/2 \) most of the time. The optimized routine is only slightly worse, and the difference is hardly noticeable. The main reason for this slightly larger average error is the smaller table size used in the optimized routine for looking up \( 2^{10} \). Note that not all library routines to compute \( e^x \) are equally accurate, and that the compiler or linker may select different routines depending on the optimization flags. If efficiency is important and an error of maybe \( 2e_m \) is acceptable, it makes sense to avoid all the overhead involved in some extra precision computations. Consequently, when we want to analyze efficiency, the reference library routine will not be the same accurate routine as above, but a faster and less accurate one. The optimized routine to compute \( e^{\pm x} \) will always be the same, however.

### 2.4.3 Optimization

We have seen the sequential algorithm to compute \( e^x \) and analyzed its accuracy until now. In this subsection we explain how to make the algorithm to compute \( e^{\pm x} \) run efficiently by applying simple optimization techniques. Of course, we will assume that our routine will be required not for a scalar value \( x \) but for a vector \( (x_k) \) of input arguments. One of the most useful first steps in optimizing is the construction of the dependence graph, which for \( e^{\pm x} \) is shown below on the left.

We may perform an operation count on the dependence graph to obtain a lower limit for the number of cycles required per loop. The graph on the left contains 12 ⧫, 14 ⧫, 7 ⧫, 2 ⧫, 8 ⧫ and 5 ⧫. For an initial estimate it is often sufficient to count the overall number of integer operations, in our case 22 ⧫. The number of cycles for a single \( e^{\pm x} \) is seen to be \( \geq 14 \) due to the 14 ⧫ required, so an optimal implementation would have to make sure that a ⧫ takes place in every cycle.
While the operation count is trivial for a given dependence graph, drawing further conclusions for the overall structure of an efficient implementation is not nearly as mechanical. In our case it is useful to estimate the latencies for \( \odot \) and derive an approximate placement in the instruction sequence, which might look like \( \odot \odot \odot \odot \odot \odot \odot \odot \odot \). It is a reasonable guess that we may avoid any gaps in the \( \odot \) sequence for the lower portion of the graph, but the initial gaps \( \odot \) and \( \odot \odot \) cannot be eliminated within a single iteration. Note that it is still possible to use software pipelining or other types of instruction scheduling to cover these gaps, but often the number of registers required increases when doing so. In this particular case, the number of registers occupied by constants like \( \frac{1}{\ln 2}, \lambda_b, \lambda_c \) and the polynomial coefficients \( \frac{1}{2}, \frac{1}{6}, \frac{1}{24}, \ldots \) for the Taylor series is quite high, so we decided to vectorize the algorithm to eliminate the gap \( \odot \odot \). In general, there is no easy way to decide if and how to vectorize a particular loop, but we mention a few advantages and disadvantages we considered for the \( e^{\pm x} \) problem.

One disadvantage of vectorization is that the loop overhead increases, since we have more than one loop now, and in some cases additional loads and stores have to be inserted. Another possible disadvantage may turn out to be an imbalance of instruction types, which results in more cycles required. In our example, after vectorization, the first phase consists of 3 \( \odot \), 4 \( \odot \) and 3 \( \odot \odot \odot \) taking 4 cycles, and the second phase consists of 9 \( \odot \), 10 \( \odot \) and 21 \( \odot \odot \odot \) taking at least 10.5 cycles. For a 4x unrolled loop the loop overhead will contribute another 0.5 cycles to the second phase, so the vectorized code requires at least 15 cycles compared to 14 cycles of the original code. We see that there is some additional cost involved in vectorization, but it may still be worthwhile or even necessary for the following reasons. The vectorized code often requires fewer registers, in our case this is obvious by looking at the constants alone, since \( \frac{1}{\ln 2}, \lambda_b, \ldots \) are needed in the first phase and \( \frac{1}{2}, \frac{1}{6}, \frac{1}{24}, \ldots \) in the second phase only. Furthermore, pipelining is often easier and requires less resources for more homogeneous codes, in our problem we could eliminate the gap \( \odot \odot \), for instance. Another advantage of this
particular vectorization is that we may essentially reuse the first phase for the computation of \((\cos x, \sin x)\) which we will look at later. For the computation \(e^{\pm x}\) we have to reduce the argument to the interval \(0 \leq x < \ln 2\) and for \((\cos x, \sin x)\) the argument will be reduced to the interval \(0 \leq x < 2\pi\), and the first phase performs exactly this argument reduction.

Of course, the initial instruction count only makes sense if there are enough registers to avoid additional loads and stores. It turns out that after vectorization the number of available registers suffices, and that we may obtain optimal implementations, optimal in the sense that the number of cycles required is given by the lower limit derived from the instruction count above. The issue maps of the sequential code and of the loop unrolled and pipelined code for the first phase are given below. Note that we assume that the number of \(e^{\pm x}\) to be computed is a multiple of 4.

The result latency of 4 cycles for \(\otimes\) and \(\otimes\) naturally suggests a \(4\times\) loop unrolling, and the structure of the dependences required a \(3\times\) overlap in software pipelining.

The second phase is not equally stringent in its dependences as the first phase, but it is still useful to use \(3\times\) pipelining to overcome eventual load latencies. As we have seen in the instruction count analysis above, the limitation in this second phase comes from the \(\otimes\) instructions together with the loop code. The issue maps of the sequential and optimized codes implementing the second phase are given below.

Again, the assumption is that \(n\), the number of \(e^{\pm x}\) to compute, is a multiple of 4.

### 2.4.4 Efficiency

In the original dependence graph, the minimal number of cycles required was found to be 14 by counting instructions. After the vectorization, the minimal number of cycles was 4.
Computing $e^x$ and 11 for the two phases, or 15 cycles overall. The issue maps given above demonstrate that the vectorized version can be implemented without further loss, at least if we make some idealizations. With a clock rate of 500 MHz, these 15 cycles correspond to 30 ns as a lower limit for computing one $e^{\pm x}$. Of course, the true execution time will be higher than 30 ns per $e^{\pm x}$ for a couple of reasons.

To analyze the efficiency of our implementation for varying vector length $n$, we use the simple model for the execution time $t = t_0 + c_1 + c_2/n$ with $t_0 = 30\, \text{ns}$. The measurement of the execution time $t$ for $e^{\pm x}$ is given below for varying vector length $n$. The execution times $t$ are given in ns per $e^{\pm x}$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$t$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>95.7</td>
</tr>
<tr>
<td>8</td>
<td>64.6</td>
</tr>
<tr>
<td>16</td>
<td>50.0</td>
</tr>
<tr>
<td>32</td>
<td>41.2</td>
</tr>
<tr>
<td>64</td>
<td>36.4</td>
</tr>
<tr>
<td>128</td>
<td>34.5</td>
</tr>
<tr>
<td>256</td>
<td>33.9</td>
</tr>
</tbody>
</table>

Fitting this data to the simple model in the least squares sense, we obtain $c_1 = 2.8\, \text{ns}$ or about 1.4 cycles overhead for each single $e^{\pm x}$, and $c_2 = 255\, \text{ns}$ overhead once for the complete vector of $e^{\pm x}$. Putting this in relation to the idealized execution time of 30 ns, we see that the execution time is about twice the optimum for $n = 8$, and the overall overhead drops to about 10% for $n \to \infty$. Taking into account that the integer pipes are highly utilized and that casual first-level cache misses will lead to partial stalls, the resulting efficiency seems satisfactory.

While the code was optimized for the Alpha 21164 microprocessor, it is still interesting to look at execution times of this piece of code on other processors for the following reasons. First, we can see that this processor dependent manual optimization does make a big difference. Second, some C compilers can be seen to fail to compile the sequential implementation of the algorithm efficiently. Third, for some processors the $e^{\pm x}$ implementation may still turn out to be faster than using $\exp$ from a standard library. An exceptional feature of the manually optimized code is that further optimizations by the compiler may actually hurt performance. We will draw a few conclusions after the following table which displays the execution times per $e^{\pm x}$ for a vector length $n = 100$. The method denoted as standard computes $e^{\pm x}$ by using $y = e^x$ and $z = 1/y$, so we do not insist on particularly accurate results.

<table>
<thead>
<tr>
<th>Processor</th>
<th>$t$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164 [500 MHz]</td>
<td>35 (gcc -O1)</td>
</tr>
<tr>
<td></td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>347</td>
</tr>
<tr>
<td></td>
<td>185</td>
</tr>
</tbody>
</table>

= optimized $e^{\pm x}$ (gcc -O1)
= optimized $e^{\pm x}$ (cc -fast)
= sequential $e^{\pm x}$ (cc -fast)
= standard $e^x$ and $1/e^x$ (cc -fast)
Although one may expect some loss of performance by letting the compiler optimize the manually optimized code, it is still surprising to see that the execution time more than doubles for the Alpha. For the MIPS it makes no big difference which option we choose, and for the UltraSPARC-II it is important to use the optimizing compiler to mitigate to some extent the lack of sufficient registers. For all processors shown in the table the optimized $e^{\pm x}$ is always more than twice as fast than the standard library approach. The efficiency of the sequential version depends significantly on the quality of the compilers used, being a factor 1.5 slower for the MIPS and a factor 4.3 slower for the Alpha compared to the optimized $e^{\pm x}$ which is optimized by the compiler.

2.5 Computing $\cos x$ and $\sin x$

In this subsection we describe how to efficiently compute pairs $\cosin x = (\cos x, \sin x)$ for a vector of values $x$. Fortunately, we may reuse some results from the earlier analysis of $e^{\pm x}$. The most important difference to $e^{\pm x}$ is that we sacrifice some accuracy in favor of efficiency, the details will be given later.

2.5.1 Algorithm

We have seen two ingredients for computing $e^x$ which we now use again for $\cos x$. For $e^x$, the remainder of $x$ with respect to $\ln 2$ had to be computed accurately to reduce the problem to the case $0 \leq x < \ln 2$. Since $\cos x$ is periodic with period $2\pi$, the same procedure with $2\pi$ instead of $\ln 2$ can be applied, being left with the case $0 \leq x < 2\pi$. We will use constants $\mu_b$ and $\mu_d$ with $\mu_b + \mu_d = 2\pi$ in the algorithm below, where the mantissa of $\mu_b$ is only a few bits wide. The other important ingredient we will reuse is the accurate representation in product form of the trigonometric functions $\cos x$ and $\sin x$ for particular $x$, denoted as

$$\cos x_t = c_t(1 + \delta_t) \quad \text{and} \quad \sin x_t = s_t(1 + \epsilon_t).$$

For small $|x|$ the trigonometric functions $\cos x$ and $\sin x$ can be efficiently computed using a couple of terms of the Taylor series

$$\cos x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k)!} x^{2k} \quad \text{and} \quad \sin x = \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1)!} x^{2k+1}.$$
Computing \( \cos x \) and \( \sin x \)

Instead of using these formulas for larger values \( x \) as well, we may use the addition theorems of trigonometric functions and use table lookup for some \( x_t \approx x \). For \( x = x_t + x_s \) with small \( x_s \) and the mantissa of \( x_t \) being only \( b \) bits wide, \( \cos x \) is given by

\[
\cos x = \cos(x_t + x_s) = \cos x_t \cos x_s - \sin x_t \sin x_s
\]

\[
\sin x = \sin(x_t + x_s) = \sin x_t \cos x_s + \cos x_t \sin x_s.
\]

Since \( x_s \) is small we have \( \cos x_s \approx 1 \), so it is a good idea not to compute \( \cos x_s \) directly, but work with \( u_s = \cos x_s - 1 \) instead. This is completely analogous to the algorithm for \( e^x \) where we had introduced \( u_s = e^{x_s} - 1 \).

We write down the algorithm for \( \cos x \) first and will discuss the underlying assumptions and the loss of accuracy later. For simplicity we assume that the number of bits \( b = 7 \) for the table lookup.

```plaintext
function \text{cosin}(x)
\begin{align*}
  y &= x \times 1/2\pi \\
  \tilde{y} &= y + 3 \times 2^{14} \\
  y_h &= \tilde{y} - 3 \times 2^{14} \\
  x_s &= (x - y_h \mu_h) - y_h \mu_s \\
  t &= (\tilde{y})_6.0 \\
  c_t(1 + \delta_t) &= \cos 2\pi(0.t) \\
  s_t(1 + \epsilon_t) &= \sin 2\pi(0.t) \\
  u_s &= -x_s^2/2 + x_s^4/24 + \cdots \\
  s_s &= x_s - x_s^3/6 + x_s^5/120 + \cdots \\
  c &= c_t + (c_t(\delta_t + u_s) - s_t s_s) \\
  s &= s_t + (s_t(\epsilon_t + u_s) + c_t s_s) \\
\end{align*}
return \((c, s)\)
```

Again, like in the algorithm to compute \( e^x \), this way of computing \( y_h \) is only possible if floating point numbers are always rounded to 53 bits in the mantissa. Note that the lookup table does not have to consist of \( 2 \times 2^b \) entries, but we may exploit identities like \( \cos(2\pi - x) = \cos x \) and \( \sin(\pi/2 + x) = \cos x \). Our implementation uses a lookup table for \( \cos x_t \) for the range \( 0 \leq x_t \leq \pi \) containing \( 2^b - 1 \) + 1 entries.

### 2.5.2 Accuracy

The above algorithm assumes that the input argument \( x \) satisfies \( |x| + \epsilon < 2\pi \times 2^{43} \) for some small margin \( \epsilon \), otherwise the argument reduction to the interval \( 0 \leq x < 2\pi \) will fail and the result may be completely wrong or even lead to a floating point exception. Depending on the splitting \( 2\pi = \mu_h + \mu_s \), or more precisely on the width of the mantissa of \( \mu_h \), argument reduction will not be accurate for all \( x \). If the mantissa of \( \mu_h \) is 21 bits wide, for instance, the product \( y_h \mu_h \) will be computed exactly for \( |x| < 2\pi \times 2^{21} \), but not for much larger \( |x| \). The computed \( \cos x \) will therefore not be particularly accurate for \( x \) outside this range.

Assuming that \( x \) is not too large, and the argument reduction step is accurate, it suffices to check the accuracy of the optimized routines for the interval \( 0 \leq x < 2\pi \). We will both look at the absolute error of \( \cos x \) for the complete interval and at the relative error of \( \sin x \) for small \( x \).
If argument reduction would be exact, \( \cos(\pi/2 + x) = \sin x \) exactly, so the analysis of \( \sin x \) for small \( x \) would actually hold for all zeros of the trigonometric functions. However, argument reduction is only accurate, but not exact, it is therefore unreasonable to expect the same small relative error for \( \cos x \) with \( x \approx \pi/2 \) compared to \( \sin x \) with \( x \approx 0 \). The following table displays the absolute error \( e_{abs} \) for the computation of \( \cos x \), comparing our optimized \( \cos x \) routine to the \( \cos \) routine from a standard math library. The \( x \) values are chosen from a uniformly distributed variable in the interval \( 0 \leq x < 2\pi \). The errors are given both in statistical notation \( e = \mu_e \pm \sigma_e \) and in graphical form.

<table>
<thead>
<tr>
<th>Optimized ( \cos x )</th>
<th>Standard ( \cos x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_{abs} = 5.3 \times 10^{-19} \pm 2.7 \times 10^{-17} )</td>
<td>( e_{abs} = 2.3 \times 10^{-19} \pm 2.7 \times 10^{-17} )</td>
</tr>
<tr>
<td>( 1 \times 10^{-16} )</td>
<td>( 1 \times 10^{-16} )</td>
</tr>
<tr>
<td>( -1 \times 10^{-16} )</td>
<td>( -1 \times 10^{-16} )</td>
</tr>
<tr>
<td>( x = 0 ) to ( x = 2\pi )</td>
<td>( x = 0 ) to ( x = 2\pi )</td>
</tr>
</tbody>
</table>

We see that the average value \( \mu_e \) is only slightly worse for our optimized routine, and that the overall error \( \sigma_e \) is about the same. Note that for \( 1/2 < \cos x \leq 1 \) the result is optimal for \( |e_{abs}| \leq \epsilon_m/4 \approx 0.55 \times 10^{-16} \). From the graphical representation we may conclude that most of the time the computed \( \cos x \) is the floating point number closest to the exact result.

The computation of \( \sin x \) for small \( x \) is special, because the relative error \( \epsilon_{rel} \) should be small, not just the absolute error \( e_{abs} \). We will analyze the sources of numerical errors and ways to eliminate them after the following table, which displays \( \epsilon_{rel} \) for the computation of \( \sin x \) in the interval \( 0 \leq x < 0.1 \). The \( x \) values are chosen from a uniformly distributed variable, and the relative errors are given both in statistical notation and in graphical form.

<table>
<thead>
<tr>
<th>Optimized ( \sin x )</th>
<th>Standard ( \sin x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \epsilon_{rel} = -6.0 \times 10^{-18} \pm 5.4 \times 10^{-17} )</td>
<td>( \epsilon_{rel} = 6.4 \times 10^{-20} \pm 4.8 \times 10^{-17} )</td>
</tr>
<tr>
<td>( 2 \times 10^{-16} )</td>
<td>( 2 \times 10^{-16} )</td>
</tr>
<tr>
<td>( -2 \times 10^{-16} )</td>
<td>( -2 \times 10^{-16} )</td>
</tr>
<tr>
<td>( x = 0 ) to ( x = 0.1 )</td>
<td>( x = 0 ) to ( x = 0.1 )</td>
</tr>
</tbody>
</table>

We can see that the average relative error for the optimized \( \sin x \) is significant, and the overall error is larger than for the \( \sin \) routines from the standard library. For an optimal routine we would expect that \( |\epsilon_{rel}| \leq \epsilon_m/2 \approx 1.11 \times 10^{-16} \) in the worst case, and the error for the \( \sin \) routine from the standard library is in fact never much larger than this. Note that the relative error for an optimal floating point result is likely to be largest for arguments \( x \) where \( \sin x = 1/2^k + \delta \) for some integer \( k \).
The accuracy of the optimized routine is not entirely satisfactory, but we will explain now where the undesired error originates from in the following formula for \( s = \sin x \) used in the algorithm

\[
s = s_t + (s_t(x_t + u_s) + c_t s_s).
\]

The optimized routine computing \( \sin x \) for which we do the error analysis uses \( b = 7 \) bits for the table lookup over the interval \( 0 \leq x < 2\pi \). Therefore the values \( \cos x_t \) are contained in the lookup table for \( x_t = 0, \pi/64, 2\pi/64, \ldots, 127\pi/64 \). For \( -\pi/128 \leq x < \pi/128 \) and therefore \( x_t = 0 \) the formula for \( s = \sin x \) used is simply

\[
s = s_s = x + (-x^3/6 + x^5/120 - \cdots),
\]

since \( s_t = 0 \) and \( c_t = 1 \), and where \( s_s \) is computed as indicated. Consequently, our optimized algorithm is reasonably accurate for the interval \( -\pi/128 \leq x < \pi/128 \). For the interval \( \pi/128 \leq x < 127\pi/128 \) with \( x_t = \pi/64 \) it can be seen in the graphics that the relative error \( \epsilon_{rel} \) is too large, particularly near the borders of the interval, so let us analyze the case \( x \approx \pi/128 \) with \( x_t = \pi/64 \). Then we have

\[
s_t(x_t + u_s) \ll -c_t s_s \approx s_t/2,
\]

which implies that the term \( s_t(x_t + u_s) \) is no problem, but that the addition \( s_t + c_t s_s \) will not be particularly accurate. To obtain more accuracy, we would have to compute \( s_s \), the product \( c_t s_s \) and the sum \( s_t + c_t s_s \) with some extra precision. Since \( s_s = x_s + v \) with \( v \ll x_s \) it is actually sufficient to compute the simpler \( s_t + c_t x_s \) more accurately. Given that \( x_t = \pi/64 \), one possibility is to split \( c_t = 1 + c_\delta \) and \( s_t = x_t + s_\delta \) for small \( c_\delta \) and \( s_\delta \). Then we may write

\[
s_t + c_t x_s = (x_t + x_s) + (s_\delta + c_\delta x_s)
\]

where the second term is relatively small and \( x = x_t + x_s \) is given exactly from the argument. Of course this rearrangement simply corresponds to computing \( \sin x \) by using \( \sin x = x + (\sin x - x) \). For larger \( x_t \) more elaborate and expensive methods may be applied to compute \( \sin x \) accurately. But we decided that the additional accuracy was not worth the associated loss in efficiency for our optimized routine to compute \( \cos x \).

2.5.3 Optimization

We now turn our attention to the efficient implementation of the algorithm to compute \( \cos x \). A simple instruction count for a sequential implementation shows that at least 15 \( \cdot \), 15 \( \cdot \) and 21 \( \\circ \circ \circ \) are required. The vectorization applied for \( e^{\pm x} \) suggests a similar strategy for \( \cos x \), especially since the argument reduction to the interval \( 0 \leq x < 2\pi \) is essentially identical to the first phase of \( e^{\pm x} \). After the vectorization, another instruction count gives 3 \( \cdot \), 4 \( \cdot \), 3 \( \\circ \circ \circ \) for the first phase and 12 \( \cdot \), 11 \( \cdot \), 20 \( \\circ \circ \circ \) for the second phase. We see that the limitation in the second phase is due to the number of multiplies, and that the minimum number of cycles required for the vectorized code is 16 compared to 15 cycles for the original code. We have mentioned earlier why vectorization may still be a reasonable thing to do.
Since the code for the argument reduction in the first phase is basically the same for $e^{\pm x}$ and \cos x, we refer to the earlier analysis. The second phase of \cos x, however, is different, and the issue maps for the sequential and the optimized code implementing it are shown below.

We see that the instructions can be scheduled such that the floating point multiply pipeline is always full, which is the limiting factor for this second phase.

### 2.5.4 Efficiency

Just as we did for $e^{\pm x}$, we measure the execution times for \cos x for varying vector sizes $n$. Again, we will use the simple model for the execution time $t = t_0 + c_1 + c_2/n$ with $t_0 = 32\, ns$.

The parameter $c_1$ measures the overhead for each \cos x independent of the vector size $n$, and $c_2$ measures the overhead appearing once for a complete vector. The execution times $t$ in the table below are given in $ns$ per \cos x.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$t$/$ns$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>104.3</td>
</tr>
<tr>
<td>8</td>
<td>70.9</td>
</tr>
<tr>
<td>16</td>
<td>55.2</td>
</tr>
<tr>
<td>32</td>
<td>44.4</td>
</tr>
<tr>
<td>64</td>
<td>38.6</td>
</tr>
<tr>
<td>128</td>
<td>36.6</td>
</tr>
<tr>
<td>256</td>
<td>36.2</td>
</tr>
</tbody>
</table>

Fitting this data to the simple model in the least squares sense, we obtain $c_1 \approx 2.9\, ns$ overhead for each \cos x, and $c_2 \approx 286\, ns$ overhead per vector. So the overhead will drop to about 9% for $n \to \infty$. This is similar to what we have seen for the optimized $e^{\pm x}$ despite the fact that the lookup table for \cos x is significantly larger, with the disadvantage of probably more load misses. Therefore, the efficiency of this implementation for computing \cos x seems reasonable.

Finally, we try our code on other processors and for different compiler flags to see how much of a difference this makes. The method denoted as \textit{standard} computes \cos x and \sin x by separate function calls to \texttt{cos} and \texttt{sin} from the math library, which is clearly not optimal but was chosen for convenience.
The conclusions we may draw from this are similar to the $e^{±x}$ case, only that the difference between compiling with gcc -01 and cc -fast is not equally large. Again, the optimized cosinx routine is much faster than the standard library approach for all processors.

2.6 Relevance

We have seen that the optimized routines for computing $f(x) = 1/\sqrt{x}, 1/x, e^{±x}$ and cosinx may be significantly faster than using standard library routines or hardware division or both, given that the vector length $n$ is large enough. The routines are even much more efficient on the Alpha 21164, the processor we have optimized the routines for. Undeniably, this efficiency is a good thing, but it also has its dark sides. We will discuss a couple of objections to the approach we have taken, and try to justify why we have nevertheless done so.

**processor specific.** A major problem with our approach is that our code will only run close to optimal on a particular processor, or maybe on a few processors implementing the same architecture. It is important to remind ourselves of the obvious fact that we do not really want to write processor specific code, but we want our algorithms to run efficiently. Currently, one prize to be paid for efficiency is that we have to optimize the code manually with a particular processor in mind. All the dependence graphs and issue maps may look nice, but we would gladly ignore all these details if compilers would be smarter. Alas, they are not, and although there is some steady progress in optimizing compilers we do not expect that manual optimizations to become completely unnecessary.

**not portable.** Another important issue is portability, or if you like the non-portability of processor specific code. Now it is true that the optimized routines for $f(x)$ run significantly faster on an Alpha 21164 than on any other processor, one reason being that the optimization is most beneficial for the Alpha. The other reason is the high clock rate of the Alpha 21164, which can only be exploited by carefully scheduled code, however. Ignoring efficiency for the moment, the implementation is still perfectly portable since it
is basically written in C, possibly interspersed with simple assembler instructions like `nop` or maybe preloads. In any case, the optimized code may be compiled and executes correctly on any processor which conforms to the IEEE 754 floating point standard. And the measured execution times show that newer microprocessors may compute even $1/x$ significantly faster using the optimized code. So the optimized code is portable, though usually not efficiently portable.

**Vectors only.** The optimized routines will only run efficiently if the vector of input arguments is large enough, and there is no speedup at all for scalar problems. Admittedly, this is fairly inflexible, and usually requires that the caller’s code be restructured. In some cases it will not even be possible for an algorithm to use the $1/\sqrt{x}$ routines in a vector-like fashion. You see, life is not always fair to algorithms, neither. On the other hand, not being able to use the optimized $1/\sqrt{x}$ routine for instance is only a problem if the computation $1/\sqrt{x}$ contributes *substantially* to the overall execution time. Naturally, an algorithm which is basically dominated by computing $1/\sqrt{x}$ will often be able to compute several $1/\sqrt{x}$ at once, unless it is restricted by data dependences. In any case, we will show in a later chapter how we structured the computation of pairwise Coulombic potentials to make efficient use of the optimized $1/\sqrt{x}$ routine.

**Hardware is faster.** Some might think that problems like $1/x$ and $1/\sqrt{x}$ should be solved in hardware. The operation $1/x$ is certainly available in hardware through floating point division. In some cases, $1/\sqrt{x}$ is implemented in hardware as well with varying degrees of accuracy. Why then should we care about solving these problems in software, which is certainly slower than hardware? For the simple reason that a software based solution may be more efficient overall, given the same amount of resources. Implementing $1/\sqrt{x}$ in hardware or making floating point division in hardware run faster will consume valuable resources. It might be a better idea to invest these resources into making basic operations like addition and multiplication faster. Admittedly, a software based solution for computing $1/\sqrt{x}$ is only efficient if several input values $x$ are available at once, while it does not solve the scalar $1/\sqrt{x}$ problem.

**Too much human work.** The amount of human work required to optimize an algorithm like $1/\sqrt{x}$ is certainly not negligible, but the simplicity of the interface at least makes bug chasing easy. The problem is simplified a lot by using only certain patterns of instruction scheduling, so that most of the tedious work like code duplication for loop unrolling can be automated. The optimized code may be generated using a simple tool written in the scripting language perl, which does the code duplication and generates pre-loop and post-loop phases for software pipelined codes, for example. Since the analysis of an algorithm and its sequential implementation, the construction of the dependence graph and some basic instruction counts are processor independent, we think that implementing $1/\sqrt{x}$ efficiently on another RISC processor than the Alpha is not especially difficult. Note however that the possible improvement is ultimately limited by the clock rate, so you should not be too disappointed if you see the Alpha still being much faster.

**Future compilers.** To close the discussion, we consider the role of compilers now and in the near future. Some might think that the time of assembly programming is over due to
the improvement of compiler technology. We do not think so. It may be true that compilers are getting smarter, but at the same time the code structure due to multi-issue architectures becomes much more sensitive to good scheduling and placement of loads and stores. Instruction scheduling is particularly important for statically scheduled processors, which achieve much higher clock rates than comparable dynamically scheduled processors. However, we think that languages should allow the programmer to express basic facts about data dependencies, the desired amount of loop unrolling or reasonable software pipelining strategies. Together with some hints on instruction scheduling, the compiler might then be able to generate reasonable code. But, even without this advanced compiler assistance, do not be afraid of manually optimizing your algorithms.

The single most important advantage of our approach to compute \( f(x) \) is that the implementations are based on floating point additions and multiplications. Unless integer operations are the limiting factor, the execution time for \( f(x) \) will decrease at the same rate as the execution time for floating point addition and multiplication. Consequently, if something can be done efficiently and with sufficient accuracy in terms of simple operations in software, it is probably a good idea to do so.
In this section we will describe the pairwise computation of the Coulombic potential and the associated forces. Given \( n \) particles with charge \( q_k \) and coordinates \((x_k, y_k, z_k)\), the mathematical formula for the potential \( \phi_i \) of the \( i \)-th particle is

\[
\phi_i = \sum_{k \neq i} q_i q_k / r_{ik},
\]

where we have ignored any constants like \( \epsilon_0 \) or \( 4\pi \), and \( r_{ik} \) is the distance between particle \( i \) and particle \( k \), that is

\[
r_{ik} = \sqrt{(x_i - x_k)^2 + (y_i - y_k)^2 + (z_i - z_k)^2}.
\]

The associated forces \( F \) are simply the derivatives of \( \phi \), so \( F_i = -\nabla \phi_i \). A pairwise computation of \( \phi_k \) and \( F_k \) for all \( k \) is characterized by the fact that the terms \( \phi_{ik} = q_i q_k / r_{ik} \) are explicitly computed for each pair of particles \( (i, k) \). In contrast, we will see in later sections how to compute \( \phi_i \) in terms of exponential functions, for instance.

Computing \( \phi \) pairwise looks deceptively simple, and a straightforward implementation seems to confirm this impression. However, an efficient implementation requires significantly more effort. In this section we will describe an implementation of \( \phi \) and analyze its efficiency.

To compute the potential \( \phi \), all possible pairs of particles \( (i, k) \) are taken into account. In many problems it suffices to consider only pairs within a certain cutoff radius \( r_{\text{cut}} \). Then the potential \( \phi_i \) of the \( i \)-th particle is given by

\[
\phi_i = \sum_{\substack{k \neq i \\ u \text{to} \ r_{ik} \leq r_{\text{cut}}}} q_i q_k / r_{ik}.
\]

We are able to partially reuse the implementation to compute \( \phi \), but some new ideas and techniques for computing \( \phi \) will be described later in this section.

### 3.1 Problem

Let us assume for the moment that we want to compute only the potentials \( \phi_i \) for all particles \( i \), but not the forces. Then a straightforward algorithm to accumulate potentials would look

```plaintext
function \( \phi \) ≡
for \( i = 1 \ldots n \) do
  for \( k = i + 1 \ldots n \) do
    \( \phi_{ik} = q_i q_k / r_{ik} \)
    \( \phi_i = \phi_i + \phi_{ik} \)
    \( \phi_k = \phi_k + \phi_{ik} \)
on
od
od
```
Unless the number of particles \( n \) is small, the above loop structure does not exploit the memory hierarchy. We see that the charge \( q_k \), the coordinates \((x_k, y_k, z_k)\) and the potential \( \phi_k \) are required for all \( i < k \). Nevertheless, the values \( q_k, \ldots, \phi_k \) will usually be thrown out of the cache between the computations of \( \phi_{ik} \) and \( \phi_{i+1,k} \), and consequently they have to be fetched from memory for each \( i \).

The data \( q_k, \ldots, \phi_k \) will remain in the cache only if the number of \( k \) particles within the inner loop is not too large. To achieve this, we split the loop over \( k = i + 4 \ldots n \) into chunks, so that the data within one chunk fits into the cache. Graphically, we may represent this split as follows

where the \( i \) particle is shown black, and the \( k \) particles fill the gray area. Note that this graphical representation does not imply that the split is related to the coordinates of the particles. However, we will see below that a subdivision based on the coordinates may be useful for another reason. Given this split, the computations may proceed in the following order

for \( i \in A \) and \( k \in B \). Therefore, the original problem of computing \( \phi_i \) for all \( i \) can be reduced to computing the Coulombic interaction \( \phi_{AB} \) for disjoint subsets \( A \) and \( B \), except for subproblems \( \phi_A \) like

when \( i \) and \( k \) particles are within the same chunk \( A \). The implementation of \( \phi_A \) is similar to the implementation of \( \phi_{AB} \), but disjoint subsets \( A \) and \( B \) occur much more often in the overall computation. Therefore, we will ignore the case \( \phi_A \) for the rest of this section.

In the following subsections, we will describe the implementation of the subproblem \( \phi_{AB} \). To avoid confusion, we introduce new indices \( a \) and \( b \) to designate particles within subsets \( A \) and \( B \). Then we want to compute

for all \( a \in A \). We have reduced the original problem \( \phi \) to many subproblems \( \phi_{AB} \) to effectively make use of the cache. Another advantage is that we may shift the subset \( A \) by a constant vector \( \lambda \) for little extra cost, as shown in the algorithm below
function $\phi_{AB} =$
for all $a \in A$ do
  $(x_a, y_a, z_a) = (x_a, y_a, z_a) + (\lambda_x, \lambda_y, \lambda_z)$ — shift by $\lambda$
for all $b \in B$ do
  $\phi_{ab} = q_a q_b / \tau_{ab}$
  $\phi_a = \phi_a + \phi_{ab}$
  $\phi_b = \phi_b + \phi_{ab}$
od

We will see in later subsections how to use this shift $\lambda$ for problems with periodic boundary conditions. In the following subsections, we will use $\phi_{AB}$ or $\phi_{\tilde{A}B}$ to denote the problem of computing the potential and the forces.

3.2 Computing $\phi_{AB}$

In this subsection we describe how to implement $\phi_{AB}$ efficiently. We start with a straightforward implementation and analyze its innermost loop to find out that the optimized routine to compute $1/\sqrt{x}$ from section 2.2 may be useful. After the code analysis, we describe the optimization steps applied, and we show some execution times for the optimized $\phi_{AB}$.

3.2.1 Naive implementation

We do not want to bore with lots of displayed C code, but it might be useful to see the real thing once, so that later transformations may be appreciated. The following piece of C code illustrates a straightforward implementation of $\phi_{AB}$. For simplicity, declarations are left out and all pointers are assumed to be properly initialized. Charges and coordinates are assumed to be arranged in tuples $x = (q, x, y, z)$ and potentials and forces in tuples $f = (\phi, f_x, f_y, f_z)$. The number of particles in subsets $A$ and $B$ is given by $n_a$ and $n_b$, and we assume no shift for $A$.

```
for (a = 0; a < na; a++) {
  qa = x_[4*a+0]; fqa = f_[4*a+0];
  xa = x_[4*a+1]; fxa = f_[4*a+1];
  ya = x_[4*a+2]; fya = f_[4*a+2];
  za = x_[4*a+3]; fza = f_[4*a+3];
for (b = 0; b < nb; b++) {
  qb = x_[4*b+0]; fqb = f_[4*b+0];
  xb = x_[4*b+1]; fxb = f_[4*b+1];
  yb = x_[4*b+2]; fyb = f_[4*b+2];
  zb = x_[4*b+3]; fzb = f_[4*b+3];
  x = xa - xb;
  y = ya - yb;
  z = za - zb;
```
\[ r = x^2 + y^2 + z^2; \]
\[ r = 1/\sqrt{r}; \]
\[ f_q = qa*qb*r; \]
\[ f_{tmp} = f_q*r*r; \]
\[ f_x = f_{tmp}*x; \]
\[ f_y = f_{tmp}*y; \]
\[ f_z = f_{tmp}*z; \]
\[ f_qa += f_q; f_qb += f_q; \]
\[ f_{xa} += f_x; f_{xb} -= f_x; \]
\[ f_{ya} += f_y; f_{yb} -= f_y; \]
\[ f_{za} += f_z; f_{zb} -= f_z; \]
\[ f[4*b+0] = fqb; \]
\[ f[4*b+1] = fxb; \]
\[ f[4*b+2] = fyb; \]
\[ f[4*b+3] = fzb; \]
\{ 
\[ f[4*a+0] = fqa; \]
\[ f[4*a+1] = fxa; \]
\[ f[4*a+2] = fya; \]
\[ f[4*a+3] = fza; \]
\}

Looking at the issue map of the innermost loop given below, we immediately recognize that there is some room for improvement.

\[ \approx 80 \]

First and most important, we see that the straightforward computation of \(1/\sqrt{r}\) alone is responsible for about \(2/3\) of the cycles. Second, there are several other stalls which cannot be eliminated within one instance of the loop body. Third, due to data dependences, we are rarely able to issue a multiplication and an addition in the same cycle.

We have seen in section 2.2 how to efficiently compute \(1/\sqrt{x}\) for a vector of \(x\) values. To use this optimized \(1/\sqrt{x}\), we have to restructure our code so that we may compute several \(1/\sqrt{x}\) at once. We have already seen that the technique of vectorization may be used for this purpose. In the dependence graph, the corresponding transformation for the inner loop looks
Note that the split into three parts is implied by our decision to use the optimized $1/\sqrt{x}$. In the dependence graph shown we assume that the values $x_0 - x_b, \ldots, z_a - z_b$ are computed in the first phase and used in the first and third phases. This reuse in the third phase requires additional load-store sequences, which may actually be more expensive than recomputing these values. The next subsection analyzes the code for the first and third phase.

3.2.2 Code analysis

In this subsection we will analyze the first and third phase of the vectorized code. The third phase is more expensive and probably harder to optimize due to its irregularities. Therefore, we analyze the third phase first, and will find out that recomputing $x_0 - x_b, \ldots, z_a - z_b$ is likely to be worse than reusing these values from the first phase. The dependence graph of the third phase is shown below.

A straightforward instruction count gives 7 $\mathbb{C}$, 8 $\mathbb{C}$, 9 $\mathbb{C}$, 4 $\mathbb{C}$ and a few $\mathbb{C}$ from the loop code. Since loads cannot be issued exactly two cycles after a store has been issued, the limiting factor for this dependence graph would be the the number of $\mathbb{C}$ and $\mathbb{S}$. However, we will see
below that we may reduce both the number of $\bullet$ and $\otimes$, so the true limiting factor are the $8 \otimes$. Therefore, it is important to avoid the $3 \otimes$ required for recomputing $x_a - x_b, \ldots, x_a - z_b$.

The following issue map for a single instance of the innermost loop does not naturally suggest an appropriate structure of the optimized code.

The gaps between the $\otimes$ speak for computing a few interactions at the same time, while several $\otimes$ may be computed simultaneously within a single instance of the loop. We might routinely loop unroll the innermost loop to exhibit the necessary instruction level parallelism. For this problem, however, it is a better idea to take into account the outer loop, as well. If we take $m_a$ particles from the outer loop and $m_b$ particles from the inner loop, we may compute the interactions of $m_a \times m_b$ pairs simultaneously. One advantage of $m_a > 1$ is that the number of loads and stores required per interaction is reduced within the innermost loop.

We have seen that $4 \times$ loop unrolling is often a good idea because of the result latency for multiplication and addition. For this third phase, it turns out that there are just enough registers to compute the interactions for $m_a = m_b = 2$, if we schedule the instructions for the 4 threads essentially in parallel. Note that the computation of these $4 = 2 \times 2$ interactions is not independent, as the following dependence graph shows.

After this $2 \times 2$ loop unrolling, the obtained larger loop body may be further software pipelined to overlap the initial multiplications and loads with later additions. This optimization step and the final issue map of the inner loop are shown below.
Next we briefly describe the efficient implementation of the first phase. We have seen that 2 × 2 loop unrolling reduces the number of loads and stores required compared to 1 × 4 loop unrolling. Another advantage is that the number of registers required for the data is proportional to \( m_a + m_b \), and obviously \( 2 + 2 < 1 + 4 \). Furthermore, the simplifying assumption that both \( n_a \) and \( n_b \) are even can be established for relatively little cost. For these reasons, we apply 2 × 2 loop unrolling to the first phase as well. Again, the 4 threads are not completely independent, as can be seen from the dependence graph of the inner loop.

There are enough registers available, and instruction scheduling is relatively easy for the 2 × 2 unrolled loop. The optimization steps involved for the inner loop are shown in the issue maps below.

Note that we have indicated the number of outer loop iterations by using the notation \( m_a \times \) or \( m_a/2	imes \).

### 3.2.3 Efficiency

In this subsection we analyze the efficiency of our implementation for \( \phi_{AB} \). For a thorough treatment, we would have to measure the execution times for several pairs of values \( n_a \) and \( n_b \). Our analysis is much less detailed and we will restrict ourselves to the special case \( n_a = 4 \) and \( n_b = 32 \).

In terms of efficiency, the choice \( n_a = 4 \) and \( n_b = 32 \) is favorable for the following reasons. First, the optimized routines expect \( n_a \) and \( n_b \) to be even. This is no problem for arbitrary \( n_a \).
since we may always add an artificial particle with charge $q = 0$. Second, for a constant number of interactions $n_a \times n_b$, the loop overhead decreases with decreasing $n_a$, so $n_a$ should be small. Third, the data which are accessed in the inner loops should fit into the cache. On the one hand, we have the charges, coordinates, potentials and forces of the $n_b$ particles. On the other hand, we have the computed $1/\sqrt{r_{ab}}$ and the reused $x_a - x_b, \ldots, z_a - z_b$. Overall this makes $8n_b + 4n_an_b$ double precision values which should fit into the 8192 bytes of the direct-mapped cache of the Alpha 21164. For $n_a = 4$ and $n_b = 32$ we have $8 \times (8n_b + 4n_an_b) = 6144 < 8192$ by a healthy margin. If we want to be very careful, we further take into account the lookup table of the $1/\sqrt{x}$ routine to avoid conflicts in the direct-mapped cache.

The above reasoning shows that we should not examine the three phases in isolation, since they are too strongly coupled. We may still estimate the contributions to the execution time of each phase by using a profiler. We compare the measured execution times to lower limits obtained from the issue maps. The first phase requires $\geq 5$ cycles, the computation of $1/\sqrt{x}$ requires $\geq 10$ cycles, and the third phase $\geq 8$ cycles. The table below summarizes optimal and measured execution times $t$ for $n_a = 4$ and $n_b = 32$, the times $t$ given in ns per interaction.

<table>
<thead>
<tr>
<th>optimal</th>
<th>first</th>
<th>$1/\sqrt{x}$</th>
<th>third</th>
<th>$\phi_{AB}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>measured</td>
<td>10.0</td>
<td>20.0</td>
<td>16.0</td>
<td>46.0</td>
</tr>
<tr>
<td></td>
<td>12.3</td>
<td>21.8</td>
<td>20.9</td>
<td>56.4</td>
</tr>
</tbody>
</table>

Note that the measured execution time for $\phi_{AB}$ includes an overhead of about 2.5% in addition to the three phases. The measured execution time for $\phi_{AB}$ is about 20% higher than the theoretical optimum, which seems satisfactory. For other $n_a$ and $n_b$ the execution may be significantly higher, particularly for small $n_b$.

The implementation of $\phi_{AB}$ optimized for the Alpha 21164 may be beneficial on other processors as well. The table below displays the execution times $t$ per interaction for a few different processors and compiler options, again using the favorable $n_a = 4$ and $n_b = 32$.

<table>
<thead>
<tr>
<th>processor</th>
<th>$t$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164 [500 MHz]</td>
<td>56</td>
</tr>
<tr>
<td>Alpha 21164 [500 MHz]</td>
<td>88</td>
</tr>
<tr>
<td>Alpha 21164 [500 MHz]</td>
<td>263</td>
</tr>
<tr>
<td>UltraSPARC-II [336 MHz]</td>
<td>212</td>
</tr>
<tr>
<td>UltraSPARC-II [336 MHz]</td>
<td>195</td>
</tr>
<tr>
<td>UltraSPARC-II [336 MHz]</td>
<td>241</td>
</tr>
<tr>
<td>MIPS R10000 [195 MHz]</td>
<td>148</td>
</tr>
<tr>
<td>MIPS R10000 [195 MHz]</td>
<td>172</td>
</tr>
<tr>
<td>MIPS R10000 [195 MHz]</td>
<td>322</td>
</tr>
</tbody>
</table>

We see that the optimized $\phi_{AB}$ on the Alpha is more than 4 times faster than the standard solution, which is partly due to the optimized $1/\sqrt{x}$ routine. Note that the overall execution time increases by more than 50% if we let the compiler optimize the manually optimized code. The difference between optimized and standard routines is small on the UltraSPARC-II. On the
other hand, the optimized $\phi_{AB}$ improves the execution time on the MIPS by more than a factor 2.

### 3.3 Computing $\phi_{AB}$

In this subsection we describe how to implement $\phi_{AB}$ efficiently, but first we show the surroundings where $\phi_{AB}$ may be usefully applied. We will see how partitioning $A \cup B \cup \ldots$ based on coordinates helps to solve the cutoff problem efficiently, for instance. The optimization of $\phi_{AB}$ itself starts with the code analysis, which suggests a natural data structure for the pairlist. Then we describe the optimization steps applied, and conclude with the analysis of the efficiency of $\phi_{AB}$. 

#### 3.3.1 Surroundings

For this subsection we assume that the $n$ particles are contained in a two-dimensional $1 \times 1$ box, which is more convenient for graphical displays than a three-dimensional box. We have already seen that partitioning the set of $n$ particles into disjoint subsets $A \cup B \cup \ldots$ is important to exploit the memory hierarchy. Let us assume that we have partitioned the large 1 $\times$ 1 box into $16 = 4 \times 4$ small boxes with associated subsets $A, B, \ldots$, then the computation of all $\phi_{AB}$ looks

![Diagram showing the partitioning of a box into smaller boxes](image)

Note that this time the boxes shown in the graphic do represent the geometric situation. For example, the first subset $A$ contains all particles $k$ with coordinates $0 \leq x_k < 1/4$ and $0 \leq y_k < 1/4$. Obviously, if the distance between two boxes $A$ and $B$ is $\geq r_{cut}$, we may ignore this pair of boxes $(A, B)$ altogether. With a cutoff radius of $r_{cut} = 1/3$ for example, the computation of all $\phi_{AB}$ could be reduced to

![Diagram showing the reduction of pairs](image)

With $4 \times 4$ boxes and $r_{cut} = 1/3$, this simple observation reduces the number of pairs $(A, B)$ from 120 to 82. Of course, the amount of reduction depends heavily on the partitioning and the cutoff radius $r_{cut}$. Given boxes $A$ and $B$ which are neither close to each other nor distant enough to be excluded on the box level, we may still exclude particles from the interaction list. The idea to reduce subsets $(A, B) \to (A', B')$ is shown graphically below

![Diagram showing the reduction of subsets](image)
Note that the elimination of a pair \((A, B)\) of distant boxes \(A\) and \(B\) may be viewed as the special case \((A, B) \to (\emptyset, \emptyset)\). These reductions are optional in the sense that we have always \(\bar{\phi}_{AB} = \bar{\phi}_{A'B'}\), since the underlying algorithm \(\bar{\phi}_{AB}\) will eliminate interactions with \(r_{ab} \geq r_{cut}\) anyway.

We have mentioned earlier that \(\bar{\phi}_{AB}\) allows the subset \(A\) to be shifted for little extra cost. Symbolically, we will write \(\bar{\phi}_{A+x,B}\) for a computation with \(A\) shifted by the vector \(x\). Since \(\bar{\phi}_{A+x,B} = \bar{\phi}_{A,B-x}\), we will never really have to shift \(B\). Shifting boxes is useful for problems with periodic boundary condition, as the following graphic illustration shows:

\[
\begin{array}{c}
\begin{array}{c}
\text{A} - (1,1) \\
\end{array} \\
\begin{array}{c}
\text{A} - (1,0) \\
\end{array} \\
\begin{array}{c}
\text{A} + (0,0) \\
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
\text{B} - (1,1) \\
\end{array} \\
\begin{array}{c}
\text{B} - (1,0) \\
\end{array} \\
\begin{array}{c}
\text{B} + (0,0) \\
\end{array}
\end{array}
\]

Let us assume that \(r_{cut} < 1\), then the pairwise potential \(\bar{\phi}_{AB}\) for boxes \(A\) and \(B\) with periodic boundary conditions may be computed as

\[
\bar{\phi}_{AB} = \bar{\phi}_{A-(1,1),B} + \bar{\phi}_{A-(1,0),B} + \cdots + \bar{\phi}_{A+(1,1),B},
\]

which reduces the problem \(\bar{\phi}_{AB}\) to the simpler problem of computing \(\bar{\phi}_{A+x,B}\) for some \(x\). Obviously, the elimination steps explained above may be equally well applied to the case \(\bar{\phi}_{A+x,B}\) for efficiency reasons.

We will not describe the efficient implementation of these ideas here, but we will use optimized routines to compute potentials \(\bar{\phi}\) for some cutoff radius \(r_{cut}\) for Ewald’s method and the PPPM method. In this section, we will restrict ourselves to the problem \(\bar{\phi}_{AB}\) from now on.

### 3.3.2 Code analysis

The straightforward implementation for \(\bar{\phi}_{AB}\) is basically the same as for \(\phi_{AB}\), except that we have to check for \(r_{ab} < r_{cut}\) for each pair of particles \(a\) and \(b\). In practice, the comparison \(r_{ab}^2 < r_{cut}^2\) will be used for efficiency reasons. For the optimized code, we again have to vectorize the routine so that \(1/\sqrt{r_{ab}}\) may be computed efficiently.

The first phase of \(\bar{\phi}_{AB}\) computes \(r_{ab}\) and eliminates all pairs \((a, b)\) with \(r_{ab} \geq r_{cut}\). Then, the second phase has to compute \(1/\sqrt{r_{ab}}\) for the subset \(S \subseteq A \times B\) of pairs \((a, b)\) which have not been eliminated before. Finally, the third phase computes potentials and forces for the same subset \(S\). Note that \(S\) does not have to be available explicitly for the second phase, but it suffices if the \(r_{ab}\) for \((a, b) \in S\) are stored contiguously in memory. The optimized routine \(1/\sqrt{x}\) will then be able to compute the vector \(1/\sqrt{r_{ab}}\) in the given order. However, the third phase needs to know the pairlist \(S\), so we have to store \(S\) in some form during the first phase. If we assume that we want to maintain the overall loop structure with an outer loop over \(a\) particles and an inner loop over \(b\) particles, \(S\) may be stored very directly as a sequence of pairs \((a, b) \in S\) like

\[
S = (1,1), (1,3), \ldots, (1,8), (2,1), (2,2), \ldots, (3,2), \ldots, (8,7).
\]

For the example above, we have assumed that \(n_a = n_b = 8\), and the ordering of pairs is implied by the loop structure. The number of the \(a\) particle is nondecreasing for the sequence \(S\), so it
suffices to know the lengths of subsequences with constant \( a \). Assuming that \( S \) in our example contains 5 pairs of type \((1, *)\), 4 of \((2, *)\), 6 of \((3, *)\) and 2 of \((8, +)\), we may write

\[
S = 5 \triangleright (1, 3, \ldots, 8), 4 \triangleright (1, 2, \ldots), 6 \triangleright (2, \ldots), 2 \triangleright (\ldots, 7).
\]

Given the above data structure for the pairlist \( S \), the algorithm for the first phase of \( \phi_{AB} \) looks

```
function first-\( \hat{\phi}_{AB} \) =
R = \{\}
S = \{\}
for \( a = 1 \ldots n_a \) do
    \( \langle x_a, y_a, z_a \rangle = x_a + \lambda \)
    \( \langle s_a \triangleright S_a \rangle = (0 \triangleright \) )
    for \( b = 1 \ldots n_b \) do
        \( \langle x_b, y_b, z_b \rangle = x_b \)
        \( \langle x, y, z \rangle = \langle x_a, y_a, z_a \rangle - \langle x_b, y_b, z_b \rangle \)
        \( r^2 = x^2 + y^2 + z^2 \)
        if \( r^2 < r_{cut}^2 \) then
            \( R = \langle R, r^2 \rangle \)
            \( \langle s_a \triangleright S_a \rangle = \langle s_a + 1 \triangleright S_a, b \rangle \)
        fi
    od
S = \{\langle S, s_a \triangleright S_a \rangle \}
od
```

Note that we have not saved the computed values \( x_a - x_b, \ldots, z_a - z_b \) as we have done in the first phase of \( \phi_{AB} \). Consequently, we will have to recompute these values in the third phase of \( \phi_{AB} \). The following analysis of the first phase will make clear why the situation is different from the case \( \phi_{AB} \).

A straightforward implementation of this first phase will contain an inner loop over \( b \) particles with an issue map similar to the following

```
\[ n_b \]
```

We know that loop unrolling and software pipelining may be used to overcome latencies induced by data dependences. The conditional branches involved in the check \( r_{ab}^2 < r_{cut}^2 \) will remain, however. Since the cost for mispredicted branches is very high, performance will be very low unless the branches are predicted correctly most of the time. Typically, this will only happen if the size of the pairlist \( |S| \) is either \( |S| \approx 0 \) or \( |S| \approx n_a n_b \). Therefore, it is important to avoid these branches, even if this involves some extra cost. For our problem, we use the following idea to eliminate the conditional branch.

\[
r^2 = x^2 + y^2 + z^2
\]
Computing $\hat{\phi}_{AB}$

$$\Delta = (r^2 < r^2_{cut}) \quad \rightarrow \Delta = 0 \text{ or } \Delta = 1$$

$R_k = r^2_{cut}$

$$k = k + \Delta \quad \rightarrow \ldots \text{ but add only for } \Delta = 1$$

After applying this transformation, we end up with an inner loop which may be routinely loop unrolled and software pipelined. The issue maps of the sequential and the optimized inner loops for this first phase are shown below.

Note that the elimination of the conditional branch requires more instructions in the code. Furthermore, all these instructions are actually executed at run time even for pairs $(a, b) \notin S$. In contrast, the original code required less instructions to be executed for this case. Nevertheless, eliminating a conditional branch is often worth some extra cost, since the execution time is the ultimate judge, and not the number of instructions.

To conclude the analysis of the first phase of $\hat{\phi}_{AB}$, we mention two important differences to $\phi_{AB}$ shown earlier. First, we do not store the computed $x_a - x_b, \ldots, z_a - z_b$ for $\hat{\phi}_{AB}$ in the first phase, since this would involve some extra cost for every pair $(a, b)$, not just for $(a, b) \in S$. Second, we use a conventional 1 x 4 loop unrolling compared to $m_a \times m_b = 2 \times 2$ loop unrolling for $\phi_{AB}$. We could use a different representation of the pairlist $S$ to enable $2 \times 2$ unrolling in the first phase, but this would not help in the third phase. Therefore, we have chosen the simpler and equally efficient solution.

The third phase of $\hat{\phi}_{AB}$ is similar to $\phi_{AB}$, except that we have to recompute $x_a - x_b, \ldots, z_a - z_b$. The dependence graph is essentially unchanged, so we will only show the issue maps of the sequential and optimized inner loops of the third phase below.

Note that the number of iteration $n'_b$ of the inner loop depends on $a$ and is given by the number of pairs $(a, *) \in S$. 
3.3.3 Efficiency

In this subsection we analyze the efficiency of our implementation of $\phi_{AB}$. Like for $\phi_{AB}$ we restrict ourselves to a special case $n_a = 8$ and $n_b = 32$ with the additional condition that exactly half of all possible pairs $(a, b) \in S$. The pairlist $S$ will therefore contain $|S| = 128$ pairs.

This choice of parameters will give us a favorable estimate of the efficiency of $\phi_{AB}$ for the following reasons. First, for a constant number of interactions $n_a \times n_b$ or $n_a \times n_b'$, the loop overhead decreases with decreasing $n_a$, so $n_a$ should be small. Second, the data accessed in the inner loops should fit into the cache. Assuming that we use 4 bytes to number particles in the pairlist, we need $8 \times 8n_b + 8 \times n_a n_b + 4 \times n_a (n_b + 1)$ bytes if every possible pair $(a, b)$ is in the pairlist $S$. We use this conservative assumption because we do not generally know in advance how many pairs $(a, b)$ can be eliminated. For our example $n_a = 8$ and $n_b = 32$ the conservative estimate is 5152 bytes, still significantly less than the 8192 bytes of the direct-mapped cache of the Alpha 21164.

Like for $\phi_{AB}$, the three phases are strongly coupled, so measuring execution times for the phases in isolation is not a good idea. However, the contributions of the three phases may be estimated by using a profiler. These estimated execution times may then be compared to the lower limits obtained from the issue maps. The first phase requires $\geq 6.5$ cycles, the computation of $1/\sqrt{r}$ requires $\geq 10$ cycles, and the third phase $\geq 11$ cycles. The table below summarizes optimal and measured execution times $t$ for $n_a = 8$ and $n_b = 32$ and $|S| = n_a n_b/2 = 128$, the times $t$ given in ns per interaction. Note that the number of interactions is $n_a n_b$ for the first phase and $|S|$ for the second and third phases.

<table>
<thead>
<tr>
<th></th>
<th>first</th>
<th>$1/\sqrt{x}$</th>
<th>third</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimal</td>
<td>13.0</td>
<td>20.0</td>
<td>22.0</td>
</tr>
<tr>
<td>measured</td>
<td>16.4</td>
<td>24.6</td>
<td>30.5</td>
</tr>
</tbody>
</table>

The measured execution times for the different phases of $\phi_{AB}$ are about 25–40% higher than the theoretical optimum, which is significantly worse than what we have seen for $\phi_{AB}$. One reason for the increased overhead is that we have to use $1 \times 4$ loop unrolling in the third phase, since we may no longer use the same $a$ particle for two $b$ particles. Note that the overall efficiency depends on $f = |S|/n_a n_b$, the fraction of pairs which belong to the pairlist $S$. For relatively small $f$, the first phase will dominate with an overhead of about 25%.

We may use the implementation of $\phi_{AB}$ on other processors as well. The table below displays the execution times $t$ per true interaction for a few different processors and compiler options, again using $n_a = 8$ and $n_b = 32$ and $|S| = n_a n_b/2 = 128$. Note that we obtain $t$ by measuring the overall execution time $T$ and dividing it by the number of true interactions, so $t = T/|S|$.

<table>
<thead>
<tr>
<th>processor</th>
<th>$t$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164 [500 MHz]</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>120</td>
</tr>
<tr>
<td></td>
<td>404</td>
</tr>
</tbody>
</table>
We see that the optimized $\phi_{AB}$ on the Alpha is more than 4 times faster than the standard solution, which is partly due to the optimized $1/\sqrt{x}$ routine. Note that the overall execution time increases by about 33% if we let the compiler optimize the manually optimized code. The difference between optimized and standard routines is small on the UltraSPARC-II. On the other hand, the optimized $\phi_{AB}$ improves the execution time on the MIPS by more than a factor 1.5. If we look at the execution times of the standard solution only, the UltraSPARC-II is fastest, followed by the Alpha and the MIPS, which are about 20% and 35% slower.

<table>
<thead>
<tr>
<th>processor</th>
<th>$t$ [n.s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>UltraSPARC-II</td>
<td></td>
</tr>
<tr>
<td>[336 MHz]</td>
<td>554</td>
</tr>
<tr>
<td></td>
<td>323</td>
</tr>
<tr>
<td></td>
<td>332</td>
</tr>
<tr>
<td>MIPS R10000</td>
<td></td>
</tr>
<tr>
<td>[195 MHz]</td>
<td>273</td>
</tr>
<tr>
<td></td>
<td>272</td>
</tr>
<tr>
<td></td>
<td>447</td>
</tr>
</tbody>
</table>

*continued*
4 MMM Theory

In this section we derive an expression for the Coulomb potential \( \phi \) in a system with periodic boundary conditions. Essentially, a series in terms of exponential functions for \( \phi \) is obtained by analytically Fourier transforming a damped Coulomb potential. We will see in later sections how to compute this series efficiently. We start with some mathematical preliminaries, then describe the problem in more detail, and finally derive a series representation for \( \phi \). Although the notation is different, the derivation given here is basically the same as the one described in [20] and [21]. We estimate the amount of work required for two different expressions for \( \phi \), and then conclude this section by summarizing important results needed in later sections.

4.1 Preliminaries

In this subsection we have collected various definitions and results which will be used for the derivation of the MMM theory. The description of the Fourier transform is useful for other methods like PPPM, as well.

4.1.1 Fourier transform

We will content ourselves with a short informal description of the Fourier transform, including its definition and some of the properties used throughout the text. The following definitions and results are taken from [3].

The Fourier transform \( F \) of a function \( f \) is defined as

\[
F(u) = \int_{-\infty}^{\infty} f(x) e^{-2\pi iux} dx.
\]

By convention, the Fourier transform of a function \( f \) will often be named \( \hat{F} \) or \( \tilde{F} \). Provided that \( f \) fulfills certain conditions, the function \( f \) can be obtained by the inverse Fourier transform of \( F \),

\[
f(x) = \int_{-\infty}^{\infty} F(u) e^{2\pi iux} du,
\]

which is basically just another Fourier transform. The following notation will be used to express the relation of one function being the Fourier transform of another.

\[ f(x) \xrightarrow{FT} F(u) \quad \text{or} \quad F(u) \xleftarrow{FT} f(x). \]

If the function \( f \) satisfies certain conditions, one can verify the following properties

\[
\begin{align*}
  f(x - \xi) &\xrightarrow{FT} e^{-2\pi i\xi u} F(u) \\
  f(\alpha x) &\xrightarrow{FT} \frac{1}{|\alpha|} F(u/\alpha) \\
  \frac{d}{dx} f(x) &\xrightarrow{FT} 2\pi iu F(u),
\end{align*}
\]
together with analogous formulas for the inverse transform. Given functions \( f \) and \( g \), the convolution product \( f * g \) is defined as

\[
f * g(x) = \int_{-\infty}^{\infty} f(x-y)g(y) \, dy.
\]

The convolution product is associative and commutative, that is

\[
(f * g) * h = f * (g * h) = f * g * h \quad \text{and} \quad f * g = g * f.
\]

In Fourier space, the convolution product corresponds to pointwise multiplication, i.e., given functions \( f \) and \( g \) with corresponding Fourier transforms \( F \) and \( G \), we have

\[
f * g(x) \xrightarrow{\text{FT}} (f * g)(u) = F(u)G(u).
\]

The shift operator as well as the differentiation operator may be written in terms of convolutions

\[
f(x - \xi) = f(x) * \delta(x - \xi)
\]

\[
\frac{d}{dx} f(x) = f(x) * \delta'(x),
\]

where \( \delta(x) \) is the Dirac distribution. Some of the basic properties of the Fourier transform become even more evident in light of the following transforms

\[
\delta(x) \xrightarrow{\text{FT}} 1
\]

\[
\delta'(x) \xrightarrow{\text{FT}} 2\pi i u.
\]

An important relation between a function \( f \) and its Fourier transform \( F \) is given by Poisson’s formula

\[
\sum_{k \in \mathbb{Z}} f(k) = \sum_{p \in \mathbb{Z}} F(p).
\]

This relation can be conveniently expressed in terms of the sampling function \( \Psi \)

\[
\Psi(x) \xrightarrow{\text{FT}} \Psi(u) \quad \text{where} \quad \Psi(x) = \sum_{k \in \mathbb{Z}} \delta(x - k).
\]

The above can be easily generalized for the expression \( \sum_{k} \delta(x - kh) \), where we have

\[
\Psi(x; h) = \sum_{k \in \mathbb{Z}} \delta(x - kh) = \Psi(x/h)/h
\]

\[
\Psi(x; h) = \Psi(x/h)/h \xrightarrow{\text{FT}} \Psi(u/h) = \Psi(u; 1/h)/h.
\]

Later in this section, we will use Poisson’s formula in the following form

\[
\sum_{k \in \mathbb{Z}} f(x + k) = \sum_{p \in \mathbb{Z}} F(p)e^{2\pi i px}.\]
4.1.2 Modified Bessel function $K_0$

The definition and properties of the modified Bessel functions given below are taken from [1]. The modified Bessel function $K_\nu(z)$ is defined to be the solution of the differential equation

$$0 = z^2 \frac{d^2 w}{dz^2} + z \frac{dw}{dz} - (z^2 + \nu^2) w$$

with the additional property $w(z) \to 0$ for $z \to \infty$. In the text, we will use the Fourier transform given in [18], formula 1.95

$$\frac{e^{-\beta \sqrt{x^2 + \rho^2}}}{\sqrt{x^2 + \rho^2}} \xrightarrow{\mathcal{F}} 2 K_0 \left( \beta \sqrt{\beta^2 + (2\pi \nu)^2} \right)$$

and its inverse

$$K_0 \left( \beta \sqrt{x^2 + z^2} \right) \xrightarrow{\mathcal{F}} \pi \frac{e^{-z \sqrt{\beta^2 + (2\pi \nu)^2}}}{\sqrt{\beta^2 + (2\pi \nu)^2}}.$$

Furthermore, we use approximations for $K_0(z)$ for both $z \to 0$ and $z \to \infty$, where $z \in \mathbb{R}$.

$$K_0(z) = -\ln z + \ln 2 - \gamma + o(z) \quad \text{for } z \to 0, \quad (4.1)$$

where $\gamma \approx 0.57722$ is Euler’s constant. Using an asymptotic expansion, we have

$$K_0(z) = \sqrt{\frac{\pi}{2z}} e^{-z} (1 + O(1/z)) \quad \text{for } z \to \infty. \quad (4.2)$$

Finally we will need the derivative of $K_0$, which is

$$K'_0(z) = -K_1(z).$$

4.1.3 Bernoulli numbers $b_n$

The Bernoulli numbers $b_n$ are defined for example by the series

$$\frac{z}{e^z - 1} = \sum_{n=0}^{\infty} \frac{b_n}{n!} z^n. \quad (4.3)$$

An identity we will need in the text is

$$\sum_{q \geq 0} \frac{e^{-qz}}{q} = -\ln(1 - e^{-z}) = -\ln z + z/2 - \sum_{n \geq 2} \frac{b_n}{n \cdot n!} z^n, \quad (4.4)$$

for $z \in \mathbb{C}$, which is valid for $\Re[z] > 0$ and $|z| < 2\pi$. The first equation in the identity is clear from the series for the logarithm.
\[
\ln(1 - z) = - \sum_{k=1}^{\infty} \frac{z^k}{k} \quad \text{for } |z| < 1,
\]

and the second equation follows from
\[
\ln(1 - e^{-z}) = \ln z + O(z)
\]
\[
\frac{d}{dz} \ln(1 - e^{-z}) = \frac{1}{e^z - 1} = \frac{1}{z} - \frac{1}{2} + \sum_{n=2}^{\infty} \frac{b_n}{n!} z^{n-1}.
\]

Note that \(b_{2n+1} = 0\) for \(n \geq 1\), so we actually have \(\sum_{n \geq 2} b_n (\cdots) = \sum_{n \geq 1} b_{2n} (\cdots)\). The convergence radius for the series in (4.3) is \(2\pi\), which implies the condition \(|z| < 2\pi\) for the series in (4.4). For later use, we note the following bounds for \(b_{2n}\) from \([1]\)
\[
\frac{2(2n)!}{(2\pi)^{2n}} < (-1)^{n+1} b_{2n} < \frac{2(2n)!}{(2\pi)^{2n}} \left( \frac{1}{1 - 2^{1-2n}} \right) \quad \text{for } n \geq 1. \tag{4.5}
\]

4.1.4 Psi functions \(\psi^{(k)}\)

Again, the definition and properties of the Psi functions are taken from \([1]\). The Digamma or Psi function \(\psi\) is defined for \(z \neq 0, -1, -2, \ldots\) as
\[
\psi(z) = \Gamma'(z)/\Gamma(z),
\]
where the Gamma function \(\Gamma(z)\) can be written in product form as
\[
1/\Gamma(z) = ze^{\gamma z} \prod_{n=1}^{\infty} (1 + z/n)e^{-z/n}.
\]

We have the expansion
\[
\psi(1 + z) = -\gamma + \sum_{n=1}^{\infty} \frac{z}{n(n + z)} = -\gamma + \sum_{n=1}^{\infty} \left( \frac{1}{n} - \frac{1}{n + z} \right),
\]
in particular \(\psi(1) = -\gamma\). The derivatives of the \(\psi\) function denoted by \(\psi^{(k)} = d^k \psi/dz^k\) may be written as
\[
\psi^{(k)}(z) = (-1)^{k+1} k! \sum_{n=0}^{\infty} \frac{1}{(n + z)^{k+1}}. \tag{4.6}
\]

From this series for \(\psi^{(k)}\), it is easy to derive an upper bound
\[
|\psi^{(k)}(z)|/k! < |1/z|^{k+1} + 1 + 1/k \quad \text{for } k \geq 1 \text{ and } z > 0.
\]
4.2 Problem

We want to compute Coulomb potentials $\phi$ for a periodic system, which can be described in terms of an elementary box $b$ of size $\lambda_x \times \lambda_y \times \lambda_z$. Equivalently, we may consider the problem of computing Coulomb potentials $\phi$ for this box $b$ with periodic boundary conditions. We assume that the box contains $n$ point charges $q_i$ and is charge neutral, that is $\sum_i q_i = 0$. The coordinates of particles within box $b$ will be denoted by $x_i = (x_i, y_i, z_i)$, where $0 \leq x_i < \lambda_x, \ldots, 0 \leq z_i < \lambda_z$.

Ultimately, we are interested in computing the potential $\phi_i$ and force $f_i$ for each particle $i$. Since we derive an analytic expression for the potential $\phi_i$, the force may be easily obtained by differentiating the final formula

$$ f_i = -q_i \nabla \phi_i. $$

Therefore, we will only consider the potential $\phi_i$ in this section.

To obtain a formula for $\phi_i$, we start with the potential energy $E$ per box, which may be written as

$$ E = \frac{1}{2} \sum_{i,j} q_i q_j \phi_{ij}, $$

where $\phi_{ij}$ is the potential for particle $i$ and all mirror images of particle $j$, including $j$ itself.

Let $r_{ij} = x_i - x_j$ be the distance vector between particles $i$ and $j$, then a more elementary expression for the energy $E$ is given by

$$ E = \frac{1}{2} \sum_{n,i,j} q_i q_j \frac{r_{ij}}{\|r_{ij} + n\|}, $$

where the sum is over all $0 \leq i, j < n$ and all mirror images of the elementary box by using shifts $n$. We use the primed sum to indicate that we omit the singular terms for $i = j$ and $n = (0, 0, 0)$. For $\sum_i q_i \neq 0$, this sum is divergent, and even for $\sum_i q_i = 0$ it is only conditionally convergent. Therefore, the computed energy $E$ depends on the order of summation. One natural order of summation is obtained by ordering terms depending on the distance of the mirror box, so we would have the following precedence relation

$$ \frac{q_i q_j}{\|r_{ij} + n\|} \prec \frac{q_i q_j}{\|r_{ij} + \tilde{n}\|} \iff \|n\| \leq \|	ilde{n}\|. $$. 

The disadvantage of this choice is that the energy $E$ then depends on the particular frame chosen for the elementary box with $n = (0, 0, 0)$. We will make the conditionally convergent sum absolutely convergent by applying an exponential damping based on the distance $\|r + n\|$. Consequently, the precedence relation in our case is given by

$$ \frac{q_i q_j}{\|r_{ij} + n\|} \prec \frac{q_i q_j}{\|r_{ij} + \tilde{n}\|} \iff \|r_{ij} + n\| \leq \|r_{ij} + \tilde{n}\|. $$

The energies $E$ computed by using the above orders of summation differ by some multiple of the dipole term $(\sum_i q_i x_i)^2$. See [16] for a discussion of this problem of conditional convergence and a derivation of the dipole term.
Since we use the distance based order of summation, the potential \( \phi_{ij} \) depends only on the distance \( r_{ij} = x_i - x_j \), so we may write

\[
\phi_{ij} = \phi(x_i, y_i, z_i, x_j, y_j, z_j) = \phi(x_i - x_j, y_i - y_j, z_i - z_j) = \phi(x, y, z).
\]

Furthermore, the function \( \phi(x, y, z) \) is periodic with periods \( \lambda_x, \lambda_y, \lambda_z \) respectively, and we have \( \phi(x, y, z) = \phi(|x|, |y|, |z|) \). Therefore, the problem of computing \( \phi_{ij} \) for arbitrary \( x_i \) and \( x_j \) reduces to the simpler problem of computing \( \phi(x, y, z) \) for \( 0 \leq x \leq \lambda_x/2, 0 \leq y \leq \lambda_y/2 \) and \( 0 \leq z \leq \lambda_z/2 \). In the following subsections, we will assume that the above inequalities hold for \( x, y \) and \( z \) without further mention.

### 4.3 Notation

To make the explanations easier to read, and easier to write above all, we introduce some notation which will be used throughout the following subsections. The specific indices \( k, l \) and \( m \) are used for sums along the \( x \), \( y \) and \( z \) coordinates, and indices \( p \) and \( q \) are used for sums in Fourier space along the transformed \( x \) and \( y \) coordinates, respectively. For mirror coordinates, we use

\[
x_k = |x + k\lambda_x|, \quad y_l = |y + l\lambda_y| \quad \text{and} \quad z_m = |z + m\lambda_z|.
\]

We use \( \rho \) to denote distances within the \( yz \) plane

\[
\rho_{lm} = \sqrt{y_l^2 + z_m^2},
\]

with possible abbreviations \( \rho_l = \rho_{l0} \) and \( \rho = \rho_0 \). The letter \( r \) is used for distances in \( xyz \) space

\[
r_{klm} = \sqrt{x_k^2 + y_l^2 + z_m^2} = \sqrt{x_k^2 + \rho_{lm}^2},
\]

again with similar abbreviations, that is, an index 0 in the last position may be left out, so \( r_{kl0} = r_{kl}, r_{k0l} = r_k \) and \( r_0 = r \).

The Fourier transforms introduce factors \( 1/\lambda_x \) and \( 1/\lambda_y \), and other terms like \( 2\pi p/\lambda_x \) will occur frequently. To get rid of this division, we let

\[
u_x = 1/\lambda_x, \quad u_y = 1/\lambda_y \quad \text{and} \quad u_z = 1/\lambda_z,
\]

or simply \( u_x = 1/\lambda_x \). Furthermore, we let

\[
u_p = 2\pi px, \quad u_q = 2\pi qx \quad \text{and} \quad u_{pq} = \sqrt{u_p^2 + u_q^2},
\]

obviously with \( u_p = u_{p0} \) and \( u_q = u_{0q} \). Note that we will use \( u_p \) and \( u_q \) only symbolically, otherwise the definition of \( u_3 \) would be ambiguous, for instance.

Finally, the damping parameter \( \beta > 0 \) will occur in certain factors after Fourier transformation, which are conveniently written as

\[
\beta_p = \sqrt{\beta^2 + u_p^2}, \quad \beta_q = \sqrt{\beta^2 + u_q^2} \quad \text{and} \quad \beta_{pq} = \sqrt{\beta^2 + u_{pq}^2}.
\]
again with $\beta_p = \beta_0 p$ and $\beta_q = \beta_0 q$ similar to above. Since $\beta > 0$, we unambiguously have $\beta_{00} = \beta_0 = \beta$ as well.

4.4 The potential $\phi$

In this subsection we derive formulas for the pairwise potential

$$\phi(x, y, z) = \sum_{k,l,m} \frac{1}{r_{klm}}.$$  

The series for $\phi$ is divergent, but we make it absolutely convergent by using exponential damping factors, that is

$$\phi(x, y, z) = \sum_{k,l,m} \frac{e^{-\beta r_{klm}}}{r_{klm}},$$

which implies that we use the distance based order of summation. We will see later that the singular terms for $\beta \rightarrow 0$ cancel due to charge neutrality. We start with a derivation of an exponential expansion of $\phi$ for the case $z > 0$. For $z \approx 0$, the expansion converges slowly, so we derive another formula for $\phi$ for the case $z \approx 0$. Finally, we let $\beta \rightarrow 0$ to obtain the final formulas for the potential $\phi$.

4.4.1 The potential $\phi$ for $z > 0$

The potential $\phi$ is given by an exponentially damped sum of electrostatic potentials

$$\phi(x, y, z) = \sum_{k,l,m} \frac{e^{-\beta r_{klm}}}{r_{klm}},$$

which is absolutely convergent for $\beta > 0$. The sums over indices $k$ and $l$ can be handled using Poisson's formula

$$f * \Pi(x; \lambda) \xrightarrow{RT} F(u)\Pi(u; 1/\lambda)/\lambda,$$

together with the Fourier transforms

$$\frac{e^{-\beta \sqrt{x^2 + \rho^2}}}{\sqrt{x^2 + \rho^2}} \xrightarrow{RT} 2 K_0 \left( \rho \sqrt{\beta^2 + (2\pi u)^2} \right)$$

$$K_0 \left( \beta \sqrt{x^2 + z^2} \right) \xrightarrow{RT} \frac{\pi e^{-z \sqrt{\beta^2 + (2\pi u)^2}}}{\sqrt{\beta^2 + (2\pi u)^2}}.$$
We obtain
\[
\phi(x, y, z) = \sum_{k,l,m} \frac{e^{-\beta_{klm}}}{r_{klm}}
= 2\pi u_x \sum_{p,l,m} K_0(\beta_{p} r_{plm}) e^{iu_p x}
= 2\pi u_x u_y \sum_{p,q,m} \frac{e^{-\beta_{pq} z_m}}{\beta_{pq}} e^{iu_q y} e^{iu_p z},
\]
noting that the term for \( p = q = 0 \) is singular for \( \beta \to 0 \). The sum \( \sum_m e^{-\beta_{pq} z_m} \) can be computed as follows. For \( \alpha = \beta_{pq} > 0 \) and temporarily using \( \lambda = \lambda_z \), we have
\[
\sum_{m} e^{-\alpha z_m} = \sum_{m < 0} e^{-\alpha [z + m\lambda]} + \sum_{m > 0} e^{-\alpha [z + m\lambda]}
= \sum_{m \geq 0} e^{-\alpha [\lambda - z]} e^{-\alpha m\lambda} + \sum_{m > 0} e^{-\alpha z} e^{-\alpha m\lambda}
= \frac{e^{-\alpha [\lambda - z]} + e^{-\alpha z}}{1 - e^{-\alpha \lambda}} = \frac{e^{\alpha z} + e^{\alpha [\lambda - z]}}{e^{\alpha \lambda} - 1}.
\]
The Laurent series of this expression around \( \alpha = 0 \), which we will use below, is given by
\[
\sum_{m} e^{-\alpha z_m} = \frac{2}{\lambda} \alpha^{-1} + \frac{\lambda^2 - 6\lambda z + 6z^2}{6\lambda} \alpha + O(\alpha^2).
\]
The Laurent series is applied to the term for \( p = q = 0 \) to extract the singularity from the sum. We apply the above formula for \( \sum_m e^{-\alpha z_m} \) and expand the singularity in terms of \( \beta \) to find
\[
\phi(x, y, z) = 2\pi u_x u_y \sum_{(p,q) \neq (0,0)} \frac{e^{\beta_{pq} z} + e^{\beta_{pq} [\lambda_z - z]}}{\beta_{pq} (e^{\beta_{pq} \lambda_z} - 1)} e^{iu_q y} e^{iu_p x} + 2\pi u_x u_y \left( u z^2 - z + \lambda_z / 6 \right) + 4\pi u_x u_y u_z \beta^{-2} + O(\beta).
\]
Note that the final formula for \( \phi \) after the elimination of \( \beta \) will be given in (4.13). For \( z \approx 0 \) the series (4.7) converges slowly, and it diverges for \( z = 0 \). Since we may permute variables \( x, y \) and \( z \), the formula is applicable if any coordinate is \( \neq 0 \) by a suitable permutation, taking the associated \( \lambda_x \) and \( u_x \) into account as well. Still, if \( x, y \) and \( z \) are all small, convergence is slow, so we derive another formula for this case.

4.4.2 The potential \( \phi \) for \( z \approx 0 \)

In this subsection we derive a formula for \( \tilde{\phi} = \phi - 1/r \) for small \( x, y \) and \( z \), which at the same time will give us the self-energy \( \phi(0, 0, 0) \). The potential \( \tilde{\phi} \) will be smooth for small \( x, y \)
and \( z \), since we have removed the interaction \( \phi_{1/r} = 1/r \) between near particles, so that only interactions of distant mirror images remain. We start by splitting the sum

\[
\bar{\phi}(x, y, z) = \sum_{(k,l,m) \neq (0,0,0)} \frac{e^{-\beta r_{klm}}}{r_{klm}} + O(\beta)
\]

into three sums \( \sum_{klm} + \sum_{kl} + \sum_{k} \) as follows

\[
\sum_{(k,l,m) \neq (0,0,0)} \frac{e^{-\beta r_{klm}}}{r_{klm}} = \sum_{k,l \neq 0} \frac{e^{-\beta r_{kl}}}{r_{kl}} + \sum_{l \neq 0} \frac{e^{-\beta r_l}}{r_l} + \sum_{k \neq 0} \frac{e^{-\beta r_k}}{r_k}.
\]

In the following subsections, we will look at each of these sums in turn. The final formula for \( \bar{\phi} \) after the elimination of \( \beta \) will be given in (4.14).

**Computing \( \sum_{klm} \)**

To compute \( \sum_{klm} \), we may proceed like in the derivation of \( \phi \). We apply Fourier transformations along \( x \) and \( y \) as before, then we are left with a sum over \( m \neq 0 \) along \( z \). Using

\[
e^{-\alpha z_m} = \frac{e^{\alpha z} + e^{-\alpha z}}{e^{\alpha z} - 1} = \frac{2}{\lambda} \alpha^{-1} - 1 + \frac{\lambda^2 + 6z^2}{6\lambda} \alpha + O(\alpha^2),
\]

we obtain

\[
\sum_{m \neq 0} \frac{e^{-\beta r_{klm}}}{r_{klm}} = 2\pi u_x u_y \sum_{(p,q) \neq (0,0)} \frac{e^{\beta pz} + e^{-\beta pz}}{\beta pq \left( e^{\beta pz} - 1 \right)} e^{iu_py} e^{iu_pz} + 4\pi u_x u_y u_z \beta^{-2} - 2\pi u_x u_y \beta^{-1} + 2\pi u_x u_y (u_z \alpha^2 + \lambda_z / 6) + O(\beta).
\]

We see that the series converges quickly for \( z \approx 0 \).

**Computing \( \sum_{kl} \)**

First, we Fourier transform \( \sum_{kl} \) along \( x \) to obtain

\[
\sum_{k \neq 0} \frac{e^{-\beta r_{kl}}}{r_{kl}} = 2u_x \sum_{l \neq 0} K_0(\beta \rho_l)e^{iu_pl} = 2u_x \sum_{l \neq 0} K_0(\beta \rho_l) + 2u_x \sum_{l \neq 0} K_0(\beta \rho_l) - 2u_x K_0(\beta \rho).
\]

The first term converges quickly for \( y \approx 0 \), so the singularity for \( \beta \to 0 \) is in the other terms. Using the estimate (4.1), we have for the third term
The potential \( \phi \)

\[
K_0(\beta \rho) = \ln 2 - \gamma - \ln(\beta \rho) + O(\beta^2). \tag{4.9}
\]

The second term requires more work. Using the Fourier transform along \( y \), we get

\[
\sum_l K_0(\beta \rho_l) = \pi u_y \sum_q \frac{e^{-\beta z}}{\beta q} e^{iu_y q} = \pi u_y \sum_{q>0} \frac{e^{-u_y q}}{u_q} (e^{iu_y q} + e^{-iu_y q}) + \pi u_y \frac{e^{-\beta z}}{\beta} + O(\beta).
\]

Letting \( \xi = 2\pi u_y (z + iy) \), and applying the identity

\[
\sum_{q>0} \frac{e^{-q \xi}}{q} = -\ln(1 - e^{-\xi}) = -\ln \xi + \xi/2 - \sum_{n \geq 2} \frac{b_n}{n \cdot n!} \xi^n
\]

given in (4.4), we obtain

\[
\sum_l K_0(\beta \rho_l) = \Re \left( -\ln(1 - e^{-\xi}) \right)_{\xi=2\pi u_y (z + iy)} + \pi u_y \beta^{-1} - \pi u_y z + O(\beta).
\]

This formula may be further simplified by noting that for \( \xi = 2\pi u_y (z + iy) \) we have

\[
\Re (-\ln \xi) = -\ln |\xi| = -\ln(2\pi u_y \rho)
\]

\[
\Re \left( \xi/2 \right) = \pi u_y z,
\]

so we find the equation

\[
\sum_l K_0(\beta \rho_l) = -\Re \left( \sum_{n \geq 2} \frac{b_n}{n \cdot n!} \xi^n \right) - \ln(2\pi u_y \rho) + \pi u_y \beta^{-1} + O(\beta). \tag{4.10}
\]

Inserting the equations (4.9) and (4.10) into the expression for \( \sum_{kl} \) yields

\[
\sum_{l \neq 0} \frac{e^{-\beta r_{kl}}}{r_{kl}} = 2u_x \sum_{\rho_l \neq 0} K_0(\beta \rho_l) e^{iu_y \rho} - 2u_x \Re \left[ \sum_{n \geq 2} \frac{b_n}{n \cdot n!} \xi^n \right]_{\xi=2\pi u_y (z + iy)}
\]

\[
+ 2u_x \gamma - 2u_x \ln(4\pi u_y) + 2\pi u_x u_y \beta^{-1} + 2u_x \ln \beta + O(\beta).
\]

As we have already mentioned for (4.4), the series in \( \xi \) has a convergence radius of \( 2\pi \). Consequently, this formula for \( \sum_{kl} \) is only applicable for \( |\xi|/2\pi = u_y \rho < 1 \).
Computing $\sum_k$

The sum $\sum_k$ can be split as follows

$$\sum_{k \neq 0} \frac{e^{-\beta r_k}}{r_k} = \sum_{k > 0} \frac{e^{-\beta r_k}}{r_k} + \sum_{k < 0} \frac{e^{-\beta r_k}}{r_k},$$

and since $x_k = |x + k\lambda_x|$ and $x_{-k} = |-(x + k\lambda_x)|$, it suffices to compute $\sum_{k > 0}$. The other sum $\sum_{k < 0}$ may then be obtained by replacing $x \to -x$. First, we isolate the term singular in $\beta$ as follows

$$\sum_{k > 0} \frac{e^{-\beta r_k}}{r_k} = \sum_{k > 0} \left( \frac{e^{-\beta r_k}}{r_k} - \frac{e^{-\beta r_k}}{k\lambda_x} \right) + \sum_{k > 0} \frac{e^{-\beta r_k}}{k\lambda_x}.$$

We observe that the first sum converges for $\beta = 0$, and for the second sum we may use

$$r_k = k\lambda_x + O(1) \quad \text{for} \quad k \to \infty.$$

Then we obtain

$$\sum_{k > 0} \frac{e^{-\beta r_k}}{r_k} = \sum_{k > 0} \left( \frac{1}{r_k} - \frac{1}{k\lambda_x} \right) + O(\beta) + e^{-\beta} \sum_{k > 0} \left( \frac{e^{-\beta k\lambda_x}}{k\lambda_x} \right)$$

$$= \sum_{k > 0} \left( \frac{1}{r_k} - \frac{1}{k\lambda_x} \right) - u_x \ln(1 - e^{-\beta \lambda_x}) e^{-O(\beta)} + O(\beta).$$

The second term may be simplified using the identity $\ln(1 - e^{-\alpha}) = \ln \alpha + O(\alpha)$ from (4.4), to find

$$u_x \ln(1 - e^{-\beta \lambda_x}) = -u_x \ln \lambda_x - u_x \ln \beta + O(\beta \ln \beta).$$

The remaining sum $\sum_{k > 0}$ may be expressed in terms of $\psi$ functions. Using the binomial series, we have for a single term

$$\frac{1}{r_k} - \frac{1}{k\lambda_x} = \frac{1}{x_k \sqrt{1 + (\beta/x_k)^2}} - \frac{1}{k\lambda_x} = \left( \frac{1}{x_k} - \frac{1}{k\lambda_x} \right) + \sum_{n > 0} \left( -\frac{1}{n} \right) \frac{\beta^{2n}}{x_k^{2n+1}}.$$

Recalling the series for the $\psi$ functions

$$\psi(1 + z) + \gamma = \sum_n \left( \frac{1}{n} - \frac{1}{n + z} \right)$$

$$\psi^{(k)}(1 + z)/k! = (-1)^{k+1} \sum_n \frac{1}{(n + z)^{k+1}} \quad \text{for} \quad k > 0,$$

we may therefore write
The potential $\phi$

$$
\sum_{k>0} \left( \frac{1}{x_k} - \frac{1}{k \lambda} \right) = -u_x (\psi(1 + u_x x) + \gamma)
$$

$$
\sum_{k>0} \frac{1}{x_k^{2n+1}} = -\frac{u_x^{2n+1}}{(2n)!} \psi^{(2n)}(1 + u_x x).
$$

Compiling the above results for $\sum_{k}$, we finally obtain

$$
\sum \frac{e^{-\beta r_k}}{r_k} = -u_x \sum_{n=0}^{\infty} \left( \frac{1}{n} \right) \frac{\psi^{(2n)}(1 + u_x x) + \psi^{(2n)}(1 - u_x x)}{(2n)!} (u_x \rho)^{2n}
$$

$$
= -2u_x \gamma + 2u_x \ln u_x - 2u_x \ln \beta + O(\beta \ln \beta). \quad (4.12)
$$

4.4.3 Letting $\beta \to 0$

In this subsection we show that the singularity in $\beta$ for the potential $\phi_t$ vanishes in the case of charge neutrality, that is $\sum q_i = 0$. The singular terms in $\beta$ for $\phi$ can be found in (4.7). For $\phi$ we have to collect singular terms from (4.8), (4.11) and (4.12). Then we have

$$
\phi(x, y, z) = 4\pi u_x u_y u_z \beta^{-2} + O(1)
$$

$$
\bar{\phi}(x, y, z) = 4\pi u_x u_y u_z \beta^{-2} - 2\pi u_x u_y \beta^{-1} + 2\pi u_x u_y \beta^{-1} + 2u_x \ln \beta - 2u_x \ln \beta + O(1)
$$

$$
= 4\pi u_x u_y u_z \beta^{-2} + O(1).
$$

We see that the singularity $S_{\beta} = 4\pi u_x u_y u_z \beta^{-2}$ is the same for $\phi$ and $\bar{\phi}$. Furthermore, $S_{\beta}$ is symmetric with respect to permutations of $x$, $y$ and $z$. Therefore, the singular term for $\phi_t$ is given by

$$
\phi_t = \sum_{j} q_j S_{\beta} + O(1).
$$

Obviously, this singular term in $\phi_t$ vanishes for $\sum q_i = 0$, so we may safely ignore singular terms and let $\beta \to 0$ in our expressions for $\phi$ and $\bar{\phi}$. Since $\beta \to 0$ implies $\beta_s \to u_x$, we obtain the following expression for $\phi$

$$
\phi(x, y, z) = 2\pi u_x u_y \sum_{(p,q) \neq (0,0)} e^{i p z} + e^{i q (\lambda_z - z)} u_{pq} (e^{i p \lambda_z} - 1) e^{i q y} e^{i u_x z}
$$

$$
+ 2\pi u_x u_y \left( u_z z^2 - z + \lambda_z / 6 \right)
$$

and the following expression for $\bar{\phi}$ from (4.8), (4.11) and (4.12)
\[
\tilde{\phi}(x, y, z) = 2\pi u_x u_y \sum_{(p,q) \neq (0,0)} \frac{e^{upqz} + e^{-upqz}}{upq(e^{upq\lambda_z} - 1)} e^{iu_y} e^{iu_px}
\]  \hspace{1cm} (4.14)

\[
+ 2u_x \sum_{p \neq 0 \atop q \neq 0} K_0(upq\rho) e^{iu_px} - 2u_x \Re \left[ \sum_{n \geq 2} \frac{b_n}{n \cdot n!} \xi^{n} \right]_{\xi = 2upq(z+i\eta)}^{}
\]

\[
+ u_x \sum_{n \geq 0} \left( \frac{1}{n} \right) \psi(2n)(1 + u_x x) + \psi(2n)(1 - u_x x) (u_x \rho)^{2n}
\]

\[
+ 2\pi u_x u_y (u_z e^2 + \lambda_z / 6) + 2u_x (\ln(u_x / u_y) - \ln(4\pi)).
\]

### 4.5 Convergence

Now we estimate the amount of work involved to compute the potentials \(\phi\) or \(\tilde{\phi}\) using expressions (4.13) or (4.14). We assume that we may neglect everything except the sums with varying numbers of terms depending on the tolerance \(\tau\). For simplicity, we will make the crude approximation that the cost for each term is about the same and simply count the number of terms. To determine the number of terms required for a sum \(\sum f(u)\), we first have to find a contour \(c\) such that \( \int f(u) \approx \tau \). Then we may count the number of terms \( f(u) \) with \( f(u) \geq c \).

Since we are particularly interested in the case where the amount of work for \(\phi\) and \(\tilde{\phi}\) is about the same, we assume further that \(z\) is not too close to 0. We use \#\(\Phi\) to denote the number of terms required to compute the series \(\Phi\), with an absolute error \(\leq \tau\).

#### 4.5.1 Convergence for \(\phi\)

With respect to the computational cost, the only interesting term from (4.13) is

\[
\phi_1 = 2\pi u_x u_y \sum_{(p,q) \neq (0,0)} \frac{e^{upqz} + e^{-upqz}}{upq(e^{upq\lambda_z} - 1)} e^{iu_y} e^{iu_px}.
\]

Since \(z \leq \lambda_z / 2\) we have \(e^{upqz} \leq e^{upq(\lambda_z - z)}\), so we may ignore the first exponential. We want to approximate the above for \(p, q \to \infty\), so \(e^{upq\lambda_z} \gg 1\). Furthermore, we take the absolute value of each term in the series. Thus we are left with

\[
\phi_1 \simeq 2\pi u_x u_y \sum_{(p,q) \neq (0,0)} \frac{e^{-upqz}}{upq}.
\]

Using new coordinates \(u = pu_x\) and \(v = qu_y\), this sum may be approximated by an integral

\[
\phi_1 \simeq \int e^{2\pi \sqrt{u^2 + v^2} / z} du dv = \int_0^\infty e^{-2\pi r z} dr d\varphi = -e^{-2\pi r z} \bigg|_0^\infty.
\]

We find \(R = -\ln(\pi z) / 2\pi z\) such that \(\int_R^\infty \cdots dr d\varphi < \tau\). The number of pairs \((p, q)\) required is then given by the number of \((p, q)\) with
\[ R^2 \geq (pu_x)^2 + (qu_y)^2. \]

Since \( R^2 = (p/\lambda_x)^2 + (q/\lambda_y)^2 \) is the equation for an ellipse with half-axes \( R\lambda_x \) and \( R\lambda_y \), we obtain

\[ \#\phi_1 \simeq \pi R^2 \lambda_x \lambda_y = \frac{\lambda_x \lambda_y \ln(\tau z)^2}{4\pi z^2}. \] (4.15)

### 4.5.2 Convergence for \( \bar{\phi} \)

In the expansion (4.14) for \( \bar{\phi} \), we count the number of terms for the four \( \sum \) and ignore the rest. Like before, we use absolute values for our estimates, and we assume that \( z \) is reasonably small.

The first sum in (4.14) is

\[ \bar{\phi}_1 = 2\pi u_x u_y \sum_{(p,q) \neq 0} \frac{e^{up_0 x} + e^{-up_0 x}}{u_p q (e^{up_0 \lambda_x} - 1)} e^{iu_p y} e^{ip_0 z}, \]

which may be handled in a similar way as \( \phi_1 \) before. By making the substitution \( z \to \lambda_x - z \) in (4.15) we obtain

\[ \#\bar{\phi}_1 \simeq \frac{\lambda_x \lambda_y \ln(\tau (\lambda_x - z))^2}{4\pi (\lambda_x - z)^2}. \]

For \( z \approx 0 \), this expression is more or less independent of \( z \) and we may write

\[ \#\bar{\phi}_1 \simeq \frac{\lambda_x \lambda_y \ln(\tau \lambda_x)^2}{4\pi \lambda_x^2} \text{ for } z \approx 0. \]

To count the number of terms of the second sum

\[ \bar{\phi}_2 = 2u_x \sum_{p, l \neq 0} K_0(u_p \rho_l)e^{ip_0 x}, \]

we use the estimate (4.2) for the modified Bessel function \( K_0 \). Since \( u_p = 2\pi pu_x \) and \( \rho_l \approx l\lambda_y \), we have

\[ \bar{\phi}_2 \simeq 8u_x \sum_{p, l > 0} K_0(2\pi u_x \lambda_y p l) + 4u_x \sum_{p > 0} K_0(2\pi u_x \rho_{-1} p). \] (4.16)

The latter sum is necessary since it may be that \( \rho_{-1} \ll \lambda_y \). The former sum may be approximated by an integral, and we want to find \( U \) such that

\[ \tau \geq 8u_x \int_{p\geq1 \atop p \geq U} K_0(\alpha pt) \, dp \, dt \text{ for } \alpha = 2\pi u_x \lambda_y. \]

Using the coordinate transform \( u = pt \) and \( v = p \) we obtain after further crude approximations
\[ \tau \geq 8u_x \int_{u \geq v \geq 1} K_0(\alpha u) v \, du \, dv \]
\[ \approx 8u_x \int_{u \geq 0} \int_{u \geq v \geq 1} v \, dv \, e^{-\alpha u} \, du \approx 8u_x \int_{u \geq 0} \frac{U^2}{2\alpha U} e^{-\alpha u}. \]

We have \( \alpha \approx 1 \) and \( U \approx 5 \) in typical cases. So a rough estimate of the value \( U \) is
\[ U \approx -\frac{\ln(\tau \lambda_x/100)}{2\pi u_x \lambda_y}. \]

For the first sum in (4.16) and \( l > 0 \) we count \( U + U/2 + U/3 + \cdots + 1 \) terms overall for \( p = 1, \ldots, U \). Using \( 1 + 1/2 + \cdots + 1/U \approx \gamma + \ln U \) and taking into account the terms \( l < 0 \) and the second sum in (4.16), we obtain
\[ \#\tilde{\phi}_2 \approx (\lambda_y/\rho - 1 + \gamma + \ln U)U \quad \text{with} \quad U \approx -\frac{\ln(\tau \lambda_x/100)}{2\pi u_x \lambda_y}. \]

The next sum we consider is
\[ \tilde{\phi}_3 = 2u_x \Re \left[ \sum_{n \geq 1} \frac{b_{2n}}{(2n)(2n)!} \xi^{2n} \right] \xi = 2\pi u_x (z + iy). \]

Again, we take absolute values and we may use the estimate \( b_{2n} \approx 2(2n)!/(2\pi)^{2n} \) from (4.5) to find an inequality for \( N \), the number of terms required
\[ \tau \geq 4u_x \sum_{n > N} \frac{(u_y \rho)^{2n}}{2n} \approx 2u_x \frac{(u_y \rho)^{2N}}{N} \frac{1 - (u_y \rho)^2}{1 - (u_y \rho)^2}. \]

Assuming that \( u_y \rho \ll 1 \) and therefore \( N \leq 5 \) in typical cases, we may neglect some factors to obtain
\[ \#\tilde{\phi}_3 \approx \ln(\tau \lambda_x)/2\ln(u_y \rho). \]

Note that the series \( \tilde{\phi}_3 \) converges only for \( u_y \rho < 1 \), which is correctly reflected in the formula for \( \#\tilde{\phi}_3 \) despite some approximations. The last sum we have to consider is
\[ \tilde{\phi}_4 = u_x \sum_{n \geq 0} \left(-\frac{1}{2}\right)^n \psi^{(2n)}(1 + u_x x) + \psi^{(2n)}(1 - u_x x)(u_x \rho)^{2n}. \]

From the series for \( \psi^{(2n)} \) given in (4.6), we may find the number \( N \) of terms required from
\[ \tau \geq u_x \sum_{n \geq N} \left(-\frac{1}{n}\right)(2 + 1/n)(u_x \rho)^{2n} + u_x \sum_{n \geq N} \left(-\frac{1}{n}\right) \frac{(u_x \rho)^{2n}}{(1 - u_x x)^{2n+1}}. \]

Assuming that \( u_x \rho \ll 1 \) and \( u_x x \ll 1 \), and neglecting some small constants, we obtain the
estimate

$$\# \tilde{\phi}_4 \simeq \frac{\ln(\tau \lambda_x)}{2 \ln(u_x \rho/(1 - u_x \lambda))}.$$ 

Again, the formula for $\# \tilde{\phi}_4$ correctly reflects the condition for convergence $u_x \rho/(1 - u_x \lambda) < 1$ despite our rough approximation.

### 4.5.3 Example

Let us make a simple example with $\# \phi_\ast$ and $\# \tilde{\phi}_\ast$ to obtain an estimate where we should switch from expression (4.13) for $\phi$ to expression (4.14) for $\tilde{\phi}$ for reasons of efficiency. Let $\lambda_x = \lambda_y = \lambda_z = 1$ and therefore $u_\ast = 1$, then we want to compute the value $z$ where both formulas require about the same amount of work, assuming that the cost is proportional to the number of terms. We let $x = y = z$ to simplify things, and let $\tau = 10^{-10}$ be the desired absolute tolerance. Note that we have four $\phi_\ast$ sums, so each of them is allowed to be wrong by $\tau/4$. Given these assumptions, the number of terms required for the cases $\phi$ and $\tilde{\phi}$ are approximately

$$\# \phi \simeq \frac{\ln(\tau z)^2}{4\pi z^2}$$

$$\# \tilde{\phi} \simeq \frac{\ln(\tau/4(1 - z))^2}{4\pi(1 - z)^2} - \left(\frac{1}{\rho - 1} + 2\right) \frac{\ln(\tau/400)}{2\pi} + \frac{\ln(\tau/4)}{2 \ln \rho} + \frac{\ln(\tau/4)}{2 \ln(\rho/(1 - z))}.$$ 

We solve $\# \phi = \# \tilde{\phi}$ numerically to obtain $z \doteq 0.39$. Quite surprisingly, the formula originally thought for small $z$ is competitive over a wide range of values. This result can be understood if we consider that the sum used in $\phi_1$ is over two dimensions. Therefore, the amount of work $\# \phi_1$ grows quadratically with $\ln \tau$, which is proportional to the number of correct digits required. Only the corresponding $\# \tilde{\phi}_4$ has the same growth rate, but with a smaller constant.

The other terms $\# \tilde{\phi}_\ast$ grow linearly with $\ln \tau$, and only the pole of $\# \tilde{\phi}_4$ for $\rho/(1 - z) = 1$ at $z = 0.41$ makes computing $\phi$ favorable.

The value $z$ for which $\# \phi = \# \tilde{\phi}$ decreases slowly with increasing $\tau$. Still assuming that $x = y = z$ and $\lambda_x = u_\ast = 1$, the cost for the two methods given a tolerance $\tau = 10^{-4}$ is about equal at $z = 0.36$, for instance.

### 4.6 Comments

Let us summarize what we will build on in later sections. We have seen a few complicated formulas, but fortunately the most efficient way to compute the periodic Coulomb potential for many particles is based on the simple formula valid for $0 < z < \lambda_x$

$$\phi(x, y, z) = 2\pi u_x u_y \sum_{(p, q) \neq (0, 0)} e^{ip_x z} + e^{ip_y (\lambda_x - z)} \frac{e^{iup_x y} e^{ip_y x}}{u_{pq}(e^{ip_q \lambda_z} - 1)} e^{ip_q \lambda_z}$$

$$+ 2\pi u_x u_y (u_x z^2 - z + \lambda_x/6)$$  \hspace{1cm} (4.17)
with $u_x = 1/\lambda_x, u_y = 1/\lambda_y, u_z = 1/\lambda_z$ and $u_p = 2\pi pu_x, u_q = 2\pi q u_y, u_{pq} = (u_p^2 + u_q^2)^{1/2}$. The number of terms $\#\phi$ required is relatively large compared to other approaches. However, this disadvantage is more than compensated by the advantage of a simple product decomposition of the following type

$$\phi(x_i - x_j, y_i - y_j, z_i - z_j) = \sum_a \chi_a(x_i, y_i, z_i) \chi_a(x_j, y_j, z_j).$$

The product decomposition for the exponential terms is simply

$$e^{a(x_i - x_j)} = e^{az} \times e^{-az}$$

$$\cos(\alpha(x_i - x_j)) = \cos(\alpha x_i) \times \cos(\alpha x_j) + \sin(\alpha x_i) \times \sin(\alpha x_j),$$

and for the polynomial terms in $z$ we have

$$(z_i - z_j)^2 = z_i^2 \times 1 - 2z_i \times z_j + 1 \times z_j^2$$

$$z_i - z_j = z_i \times 1 - 1 \times z_j$$

$$1 = 1 \times 1.$$

Another important observation is that the estimate for the number of terms required

$$\#\phi \approx \frac{\lambda_x \lambda_y \ln(z^2)}{4\pi \frac{z^2}{z^2}}$$

depends not only on $z$, but on the box dimensions $\lambda_x$ and $\lambda_y$ in the other directions as well. For $0 < z \leq \lambda_z/2$ we see that the cost decreases for increasing $z$ and decreasing $\lambda_x$ and $\lambda_y$. We will exploit this fact later by artificially making some $\lambda_z$ smaller with additional mirror images.
5 Other m body methods

In this section we briefly describe other methods than MMM to compute Coulomb interactions, not necessarily with periodic boundary conditions, though. We start with a description of the ancient Ewald sum [9], which may still be useful for some problems. In section 8 we will compare the execution times of Ewald's method and MMM for small problems. The next item is the PPPM method, which is described in [8] or [13], for instance. The PPPM method is applicable to problems where the required accuracy is relatively low. We will compare the execution times of PPPM and MMM for varying error tolerance $\tau$ in section 8. Both Ewald's method and PPPM naturally solve the Coulombic problem with periodic boundary conditions. We use the description of FMM, the fast multipole method, for an introduction to the concept of product decomposition. Since they will be used later, we describe ideas like shifting or converting expansions in detail. Finally, we mention a variant of FMM based on an exponential expansion of $1/r$. Although FMM can be used to solve the periodic problem, it is more directly applicable to the non-periodic problem. For an introduction to FMM and further references see [12].

5.1 Ewald's method

The basic idea in deriving Ewald's formula is splitting the potential
\[ \phi = 1/r = \phi_s(r) + \phi_l(r) \]
into a short range part $\phi_s$ with a singularity at $r = 0$ and a smooth long range part $\phi_l$, which can be Fourier transformed analytically. Classically, the parts $\phi_s$ and $\phi_l$ are given by
\[ \phi_s(r) = \frac{1}{r} - \frac{\text{erf}(\alpha r)}{r} = \frac{\text{erfc}(\alpha r)}{r} \]
\[ \phi_l(r) = \frac{\text{erf}(\alpha r)}{r} \]
for some parameter $\alpha$, where erf is the error function defined by
\[ \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt. \]
The potential $\phi_l(r)$ corresponds to a Gaussian shaped charge distribution
\[ \sigma(r) = \frac{4\alpha^3}{\sqrt{\pi}} e^{-\alpha^2 r^2}, \]
which can be shown as follows. For the radial function $\phi_l(r)$, we have
\[ \Delta \phi_l(r) = \frac{1}{r} \frac{\partial^2}{\partial r^2} (r \phi_l(r)) = -\frac{1}{\sqrt{\pi}} \frac{\partial}{\partial r} e^{-\alpha^2 r^2} = -\frac{4\alpha^3}{\sqrt{\pi}} e^{-\alpha^2 r^2} = -\sigma(r) \]
as required by one of Maxwell’s equations, if we ignore the constant $4\pi \epsilon_0$.

Let us assume that the elementary box $b$ has size $1 \times 1 \times 1$, then the long range potential $\phi_l$ from all mirror images is given by $\phi_l = \phi_l \ast \mathbb{M}^3$, where $\mathbb{M}^3$ represents the sampling function in three dimensions. Using the Fourier transform

$$e^{-\pi x^2} \xrightarrow{\text{FT}} e^{-\pi u^2},$$

we may compute the Fourier transform of $\phi$ in three dimensions

$$e^{-\alpha^2 \tau^2} \xrightarrow{\text{FT}^3} \left(\frac{\sqrt{\pi}}{\alpha}\right)^3 e^{-\pi^2 u^2/\alpha^2},$$

$$\sigma(\tau) \xrightarrow{\text{FT}^3} \hat{\sigma}(u) = 4\pi e^{-\pi^2 u^2/\alpha^2}.$$

Given the Fourier transform of $\sigma$, the transform of $\phi_l$ may then be computed by solving the Laplace equation in Fourier space

$$-4\pi u^2 \hat{\phi}_l = \hat{\sigma}.$$

Consequently, we obtain the following series for the periodic long range potential

$$\phi_l(x, y, z) = 4\pi \sum_{p,q,r} \frac{u_{pqr}^2/4\alpha^2}{u_{pqr}^2} e^{2\pi i px} e^{2\pi i qx} e^{2\pi i rx} \quad (5.1)$$

where $u_{pqr} = 2\pi \sqrt{p^2 + q^2 + r^2}$. In the sum above, the singular term for $p = q = r = 0$ can be ignored due to charge neutrality. The singularities due to the self-interaction terms $q_i^2/2r_i^2$ have to be eliminated as follows. We omit the short range interaction $\phi_s$ for pairs $(i, k)$ with $i = k$ and add corrections $-\alpha q_i^2/\sqrt{\pi}$, since $\lim_{r \to 0} \phi_s(r) = 1/r - 2\alpha/\sqrt{\pi}$.

The series (5.1) has a simple product decomposition in terms of cos and sin, so the amount of work for $n$ particles is only $O(n)$ instead of $O(n^2)$ for constant $\alpha$. Since the short range part is $O(n^2)$ for constant $\alpha$, the method is still $O(n^2)$ overall. If $\alpha$ is allowed to vary with increasing $n$, then the amount of work for Ewald’s method can be made $O(n^{3/2})$ by choosing $\alpha \sim n^{1/6}$, since the amount of work is $O(n^2/\alpha^3)$ for $\phi_s$ and $O(n\alpha^3)$ for $\phi_l$.

### 5.2 PPPM method

The name PPPM is an acronym for particle-particle particle-mesh. Like in Ewald’s method, the potential $\phi$ is split into a short range part $\phi_s$ and a smooth long range part $\phi_l$. The short range potential $\phi_s$ is computed pairwise (particle-particle), and the periodic long range potential $\phi_l$ is computed via Fourier transform on a discrete mesh (particle-mesh). The time complexity of PPPM is $O(n)$ for the pairwise computation of $\phi_s$ and $O(n \log n)$ for the computation of $\phi_l$ via Fourier transform, if we assume that the number of mesh cells grows linearly with $n$.

The Fourier transform in PPPM is computed numerically compared to the analytic Fourier transform in Ewald’s method. Consequently, we have more freedom in choosing an appropriate long range potential $\phi_l$. However, it turns out that the long range potential $\phi_l$ used in Ewald’s
method is about as good as other possible \( \phi_t \). Therefore, our implementation of PPPM uses the same splitting

\[
\phi_x(r) = \frac{1}{r} \text{erf}(\alpha r) = \frac{\text{erfc}(\alpha r)}{r}
\]

\[
\phi_t(r) = \frac{\text{erf}(\alpha r)}{r},
\]

with the advantage that we may use the same routine to compute \( \phi_x \) for both Ewald’s method and PPPM.

For the following description of the algorithm, we assume that the elementary box \( b \) is of size \( 1 \times 1 \times 1 \), and that we use a \( m \times m \times m \) mesh. The size of a mesh cell is then \( h \times h \times h \) with \( h = 1/m \), and we will use the index \( p \) to indicate mesh points. We assume further that we want to compute forces \( f \) for \( n \) particles with charges \( q_i \) and coordinates \( x_i \). Then, the PPPM algorithm to compute the particle forces for \( \psi = \phi_t \) proceeds in the following steps.

(i) Assign the charges from the particles to the mesh points using inverse interpolation

\[
\rho_p = \sum_i q_i w(x_i - x_p)
\]

for a given interpolation function \( w \). In our implementation we use the third-order product scheme TSC given by

\[
w(x) = \frac{3}{4} - \frac{x^2}{2} \quad 0 \leq |x| < \frac{1}{2}
\]

\[
p(x) = \begin{cases} 
\frac{3}{4} - \frac{x^2}{2} & 0 \leq |x| < \frac{1}{2} \\
\frac{3}{2} - |x|^2 & \frac{1}{2} \leq |x| < \frac{3}{2} \\
0 & \text{otherwise}
\end{cases}
\]

(ii) Compute the Fourier coefficients \( \rho_u \) by using the discrete Fourier transform on the mesh

\[
\rho_p \xrightarrow{\text{DFT}} \rho_u.
\]

(iii) Compute the Fourier coefficients \( \psi_u \) using

\[
\psi_u = G_u \rho_u,
\]

where the coefficients \( G_u \) are determined to minimize the error in the force. More precisely, the following expression \( Q \) is minimized

\[
Q = \int \int \| f(x; y) - f_\psi(x - y) \|^2 \, dx \, dy \rightarrow \text{min},
\]

where \( f(x; y) \) is the force computed by PPPM for two particles at \( x \) and \( y \), and \( f_\psi \) is the exact force associated to \( \psi \).
(iv) Use the inverse discrete Fourier transform to determine the potential $\psi_p$ at mesh points

$$\psi_p \xrightarrow{\text{DFT}} \psi_u$$

and apply a finite difference operator $D$ to obtain the field $E_p$. In our implementation we use the simple two-point difference operator to obtain

$$E_p = -(\psi_{p+h} - \psi_{p-h})/(2h).$$

(v) Finally, the forces $f_i$ for particles are found by interpolation with the same interpolation function $w(x)$ used earlier

$$f_i = g_i \sum_p w(x_i - x_p)E_p.$$ 

Note that the parameters $G_u$ depend only on the given box $b$ and its discretization $m \times m \times m$, the chosen interpolation function $w$, the force $f_u$ and the finite difference operator $D$. Therefore, the parameters $G_u$ have to be computed only once per simulation.

5.3 FMM — Fast multipole method

In this section we describe the basic ideas of FMM, the fast multipole method, which has a time complexity of $O(n)$ or $O(n \log n)$ depending on the particular implementation. The FMM is based on the following two ingredients. First, the time complexity of certain subproblems of size $m$ can be reduced from $O(m^2)$ to $O(m)$ by using a product decomposition. Second, these subproblems can be combined efficiently by shifting and maybe converting the expansions associated to the product decomposition. Although the description of these ideas is given in the context of FMM, the description of MMM given later will partly depend on this subsection. For an introduction to FMM and further references see [12].

5.3.1 Product decomposition

Assume we want to compute

$$\phi = \sum_{0 \leq i, j < n} f(x_i, x_j)$$

for some function $f$ and given points $x_i$. A direct computation of $\phi$ requires $n^2$ evaluations of the function $f$, so the amount of work is $O(n^2)$. If the function $f$ has a product decomposition

$$f(x, y) = \sum_{p=1}^{P} \tilde{\chi}_p(x)\chi_p(y),$$

then $\phi$ may be computed as

$$\phi = \sum_{0 \leq i, j < n} \tilde{\chi}_p(x_i)\chi_p(x_j) = \sum_{1 \leq p \leq P} \left( \sum_{0 \leq i < n} \tilde{\chi}_p(x_i) \right) \left( \sum_{0 \leq j < n} \chi_p(x_j) \right),$$
the amount of work now being $O(2nP)$. Therefore, if we are only interested in $n$, the time
complexity to compute $\phi$ reduces to $O(n)$. It is often sufficient to approximate $f$ by a product
decomposition

$$f(x, y) \approx \sum_{p=1}^{P_r} \chi_p(x) \chi_p(y),$$

where the number of terms $P_r$ required typically depends on the error tolerance $\tau$. In some
cases, the decomposition functions $\chi_p$ and $\chi_p$ may depend on $\tau$ as well.

To compute the Coulomb potential efficiently, we are interested in a product decomposition of
the function $1/\|x - y\|$. Let $x = (r, \theta, \varphi)$ and $y = (\rho, \alpha, \beta)$ be the spherical coordinates of
two points $x$ and $y$ with $\rho < r$. Furthermore, let $\gamma = \angle(x, y)$ be the angle between the rays
from the origin to $x$ and $y$, and let $R = \|x - y\|$ be the distance between the two points. Using

$$R^2 = r^2 + \rho^2 - 2r\rho \cos \gamma$$

$$\cos \gamma = \cos \theta \cos \alpha + \sin \theta \sin \alpha \cos(\varphi - \beta),$$

we have

$$\frac{1}{R} = \frac{1}{r \sqrt{1 - 2(\rho/r) \cos \gamma + (\rho/r)^2}} = \sum_{n=0}^{\infty} \frac{\rho^n}{r^{n+1}} P_n(\cos \gamma),$$

where $P_n$ are the Legendre polynomials. This is not enough yet for a complete product decom¬
position, since the angles $\theta, \varphi$ and $\alpha, \beta$ are still intertwined. Their separation is accomplished
using the addition theorem for Legendre polynomials

$$P_n(\cos \gamma) = \sum_{m=-n}^{n} Y_n^{-m}(\alpha, \beta) Y_n^m(\theta, \varphi),$$

where $Y_n^m$ are the surface harmonics of the first kind, see [1]. The product decomposition of
$1/R$ is then given by

$$\frac{1}{R} = \sum_{n,m} Y_n^{-m}(\alpha, \beta) \rho^n \frac{Y_n^m(\theta, \varphi)}{r^{n+1}} \text{ for } \rho < r. \tag{5.2}$$

We will see in the next subsection how to exploit this decomposition.

5.3.2 Translation and conversion

The product decomposition of $1/R$ given in (5.2) may be used to compute the interaction
between well-separated boxes, where each box may contain several particles. An illustration
of the situation is given below
Note that we must have \( p_i < r_j \) for every pair \((i, j)\) of particles, so we cannot simply pick arbitrary subsets. Assuming unit charges \( q_i = q_j = 1 \), the potential \( \phi \) for a single \( r \)-particle is given by the multipole expansion

\[
\phi = \sum_{n,m} M_n^m \frac{Y_n^m(\theta, \varphi)}{r^{n+1}}
\]

with \( M_n^m = \sum_i Y_{n-m}^m(\alpha_i, \beta_i) \rho_i^m \) being computed from the particles in the shaded box. There is some freedom in choosing the origin for the spherical coordinates. An implementation may use the center of the shaded box for practical reasons, or it might try to minimize \( \rho / r \) such that the number of terms for the approximate decomposition is minimized.

If we can shift the \( \rho \)-origin, then several shaded boxes can be combined into a single multipole expansion for the \( r \)-particle as shown in the sketch below.

Omitting the details in the coefficient transform \( M_n^m \rightarrow \bar{M}_n^m \), a shift of the \( \rho \)-origin transforms the multipole expansion for \( \phi \) to

\[
\bar{M}_n^m = \sum_{j=0}^{n} \sum_{k=-j}^{j} (\cdots) M_j^k.
\]

If the location of the \( \rho \)-origin could be freely chosen, the ability to shift an expansion would be sufficient. In our case, we are restricted by the condition \( \rho < r \) for the decomposition. Therefore, we need the conversion from a multipole to a so-called local expansion accompanied by the coordinate transform \( \bar{\rho} \rightarrow \rho \) as shown below.
The new local expansion and the formula for the coefficient transform $M^m_n \rightarrow L^m_n$ are then given by

$$
\phi = \sum_{n,m} \tilde{L}^m_n Y^m_n(\tilde{\alpha}, \tilde{\beta}) \tilde{\rho}^n
$$

$$
\tilde{L}^m_n = \sum_{j=0}^{\infty} \sum_{k=-j}^{j} \cdots \tilde{M}^k_j.
$$

Typically, a local expansion with a single $\tilde{\rho}$-origin is valid for several small boxes. However, it may be more appropriate to have a different $\tilde{\rho}$-origin for each small box. Consequently, it is useful if we can shift the $\tilde{\rho}$-origin of local expansions, as well. This shift and the associated coordinate transform $\tilde{\rho} \rightarrow \rho$ is shown below:

The new local expansion and the coefficient transform $\tilde{L}^m_n \rightarrow \tilde{L}^m_n$ are given by

$$
\phi = \sum_{n,m} L^m_n Y^m_n(\alpha, \beta) \rho^n
$$

$$
L^m_n = \sum_{j=0}^{\infty} \sum_{k=-j}^{j} \cdots \tilde{L}^k_j.
$$

We have seen that the product decomposition (5.2) requires shifts and maybe conversions of expansions. If we consider only a pair of boxes, the process essentially reduces to a sequence of coefficient transforms $M^m_n \rightarrow M^m_n \rightarrow \tilde{L}^m_n \rightarrow \tilde{L}^m_n$. This route from $M^m_n \rightarrow \tilde{L}^m_n$ is not necessarily the most efficient, since shifts and particularly conversions are relatively expensive in this representation. An alternative is to use an approximation by an exponential expansion

$$
\phi = \sum_{n,m} M^m_n Y^m_n(\theta, \psi) e^{\lambda_k x e^{i\lambda_k (x \cos \alpha_j + y \sin \alpha_j)}},
$$

and an analogous formula for the local expansion. Then the process can be written $M^m_n \rightarrow W^k_j \rightarrow L^m_n$, if we ignore the shift, which is inexpensive in the exponential representation $W^k_j$. Note that the origin may be freely chosen in the exponential expansion, so no conversion is needed.

In the complete FMM algorithm, shifts and conversions are used to combine multipole expansions and to split local expansions hierarchically. We will not describe this process for the original FMM algorithm, but we will explain the problems involved in the context of exponential expansions in the following subsections.
5.3.3 Exponential expansion

As we have seen above, exponential expansions may be useful as intermediate representations, since they are easy to translate. It is even possible to use the exponential expansion directly to compute Coulomb potentials without intervening multipole or local expansions. The following integral representation from [17] may be used to obtain exponential expansions

\[
\frac{1}{R} = \frac{1}{\sqrt{x^2 + y^2 + z^2}} = \frac{1}{2\pi} \int_0^\infty e^{-\lambda z} \int_0^{2\pi} e^{i\lambda (z \cos \alpha + y \sin \alpha)} d\alpha d\lambda, \tag{5.3}
\]

which is valid for \( z > 0 \). It may be derived for example using the simpler formulas

\[
\frac{1}{\sqrt{1 + z^2}} = \int_0^\infty e^{-\lambda z} J_0(\lambda) d\lambda \\
J_0(z) = \frac{1}{\pi} \int_0^{\pi} e^{iz \cos \theta} d\theta
\]

given in [1]. The inner integral in (5.3) can be approximated by the trapezoidal rule, and the integration with respect to \( \lambda \) may be approximated with generalized Gaussian quadrature rules, see [22]. By approximating the integral, we obtain an approximate expansion valid for \( z > 0 \)

\[
\frac{1}{\sqrt{x^2 + y^2 + z^2}} \approx \sum_k w_k e^{-\lambda_k z} \sum_{j=0}^{M_k-1} e^{i\lambda_k (z \cos \alpha_{jk} + y \sin \alpha_{jk})}, \tag{5.4}
\]

where \( \alpha_{jk} = 2\pi j / M_k \), and the weights \( w_k \) and nodes \( \lambda_k \) are given by the generalized Gaussian quadrature rule. Of course, the number of terms required in the expansion (5.4) increases with decreasing error tolerance \( \tau \) and decreasing \( z \).

In principle, we have explained earlier how product decomposition can be used to compute the potential \( \phi \) for a pair of boxes, assuming that \( z > 0 \) for all pairs of particles. However, since MMM uses an exponential expansion, we explicitly write down its product decomposition and describe the computational steps in detail. The terms in sum (5.4) belonging to different \( \lambda_k \) may be handled independently, and constant factors do not pose a problem, so we will only consider sums like

\[
\phi_\lambda = e^{-\lambda z} \sum_{j=0}^{M-1} e^{i\lambda (c_j x + s_j y)},
\]

where we have set \( c_j = \cos(2\pi j / M) \) and \( s_j = \sin(2\pi j / M) \). For simplicity, let us assume that \( M \) is a multiple of 4, then we have

\[
c_0 = 1, \ c_{M/4} = 0, \ c_{M/2} = -1, \ c_{3M/4} = 0 \\
s_0 = 0, \ s_{M/4} = 1, \ s_{M/2} = 0, \ s_{3M/4} = -1
\]

and furthermore

\[
c_j = -c_{M/2-j} = -c_{M/2+j} = c_{M-j} \\
s_j = s_{M/2-j} = -s_{M/2+j} = -s_{M-j}
\]
for \(0 < j < M/4\). This reduces the number of terms for \(\phi_\lambda\) to about \(M/4\), and at the same time we get rid of the complex exponential to obtain

\[
\phi_\lambda = e^{-\lambda z} \left( 2 \cos(\lambda x) + 2 \cos(\lambda y) + 4 \sum_{j=1}^{M/4-1} \cos(\lambda c_j x) \cos(\lambda s_j y) \right).
\]

To illustrate the steps to compute \(\phi_\lambda\) using product decomposition, we pick the term for \(j = 1\) from the sum above. We let \(c = c_1\) and \(s = s_1\), ignore constant factors and use \((x, y, z) = \mathbf{x} - \mathbf{x}\) for two particles at \(\mathbf{x}\) and \(\mathbf{x}\). Assuming that \(z > 0\), we have

\[
\phi_1 = e^{-\lambda(z-z)} \cos(\lambda c(\mathbf{x} - \mathbf{x})) \cos(\lambda s(\mathbf{y} - \mathbf{y})).
\]

Then the product decomposition is obtained from the addition theorem for trigonometric functions

\[
\phi_1 = e^{-\lambda z} \cos(\lambda c\mathbf{x}) \cos(\lambda s\mathbf{y}) \times e^{\lambda z} \cos(\lambda c\mathbf{x}) \cos(\lambda s\mathbf{y})
+ e^{-\lambda z} \cos(\lambda c\mathbf{x}) \sin(\lambda s\mathbf{y}) \times e^{\lambda z} \cos(\lambda c\mathbf{x}) \sin(\lambda s\mathbf{y})
+ e^{-\lambda z} \sin(\lambda c\mathbf{x}) \cos(\lambda s\mathbf{y}) \times e^{\lambda z} \sin(\lambda c\mathbf{x}) \cos(\lambda s\mathbf{y})
+ e^{-\lambda z} \sin(\lambda c\mathbf{x}) \sin(\lambda s\mathbf{y}) \times e^{\lambda z} \sin(\lambda c\mathbf{x}) \sin(\lambda s\mathbf{y}).
\]

The decomposition for \(z < z\) is obtained by replacing \(e^{-\lambda z}\) by \(e^{\lambda z}\) and \(e^{\lambda z}\) by \(e^{-\lambda z}\). Note that the values for \(\cos(\cdots)\) and \(\sin(\cdots)\) remain unchanged for \(z \to -z\) and can therefore be used for both cases \(z > 0\) and \(z < 0\).

5.3.4 Combining boxes

Given the above product decomposition, we may efficiently compute \(\phi\) for a pair of boxes. In this subsection, we describe the computation on the box level. The expansion given in (5.4) is only valid for \(z > 0\), but analogous formulas exist for \(z < 0\) or along other coordinates \(x\) and \(y\). Therefore, we are able to efficiently compute the potential \(\phi_{bb}\) for those pairs of boxes \((b, \tilde{b})\) where the distance between \(b\) and \(\tilde{b}\) is not too small. To illustrate the combining process on the box level, we assume that all particles are contained in a two-dimensional box of size \(1 \times 1\), which is partitioned into \(12 \times 12\) small boxes. The sketch below shows a black box \(b\) and the associated gray region of \(\tilde{b}\) boxes where \(\phi_{bb}\) is computed using the expansion for \(z > 0\).

![Sketch of boxes](image)

By changing the sign \(z \to -z\) or by using permutations like \(x \to z \to x\), we may compute contributions from other directions as well. In our example, the combined gray region of \(\tilde{b}\) boxes at this level is shown below
The contributions of boxes \( b \) far away from \( b \) can be computed in an analogous way on a coarser \( 6 \times 6 \) grid. The contributions from both levels may then be combined to compute \( \phi_b \) as shown in the illustration below.

The expansion for this coarser grid can be obtained by scaling the coordinates \( x, y \) and \( z \) by a factor 2 in the original formula (5.4). With \( z \rightarrow 2z \) we have \( e^{\lambda x} \rightarrow e^{2\lambda x} = (e^{\lambda x})^2 \), and similar relations exist for terms in \( x \) and \( y \) due to \( x \rightarrow 2x \) and \( y \rightarrow 2y \). Consequently, computing values for different levels may be implemented more efficiently than solving the problem for each level in isolation. We will see in a later section how to exploit this observation.

So far, we have seen how to compute the potential \( \phi_b \) for a single black box \( b \), given an interaction list of gray boxes \( \tilde{b} \) as shown below.

Instead of combining interaction lists for each black box \( b \) in isolation, gray boxes may be combined in a hierarchical way. The interaction list for a particular black box \( b \) will then be partitioned into subregions, which can be reused by other black boxes. We explain the idea with an example, where we combine gray boxes into subregions containing 6 gray boxes each. Then we can compute \( \phi_b \) for 24 black boxes \( b \) with only 12 gray subregions required, as illustrated below.

In some situations it makes sense to compute contributions not for a single black box \( b \), but for groups of black boxes. In our case, a simple optimization would be to compute the contribution for pairs of black boxes as follows.
Note that subregions are only important for reasons of efficiency. Whatever scheme of combination is used, we always have to compute the contributions from all grey boxes $\tilde{b}$ in the interaction list of the black box $b$. 
6 MMM Algorithm

This section describes the MMM algorithm based on the MMM theory given in section 4. The problem we want to solve is the computation of the Coulomb potential $\phi$ for periodic boundary conditions. First, we split this problem into two subproblems, which compute the potential $\phi$ in the far range and the potential $\phi = \phi_1 + \phi_1/r$ in the near range, respectively. The overall structure of these subproblems is similar, so we describe in detail only the case for the far range $\phi$.

6.1 Notation

We assume that we are given $n$ particles with charge $q_i$ and coordinates $x_i = (x_i, y_i, z_i)$, which are contained in a box of size $\lambda x \times \lambda y \times \lambda z$. We will use graphical illustrations to display Coulomb interactions between subregions of this box. We introduce the convention of using a black box $b$ and a grey box $\bar{b}$ to depict the computation of the potential $\phi_b$ for the black box $b$ as follows.

![Graphical Illustration](image)

In more general cases, we use black regions $c$ and grey regions $\bar{c}$ to indicate the computation of the potential $\phi_c$.

Although we are interested in the three-dimensional problem, we will introduce the two-dimensional potential $\psi(x, y)$ below to explain part of the algorithm. In the two-dimensional case, we assume that there are $n$ particles with charge $q_i$ and coordinates $x_i = (x_i, y_i)$, which are contained in a box of size $\lambda x \times \lambda y$.

6.2 Subproblems

We use the expression (4.17) for the potential $\phi$ derived in section 4, and since the polynomial terms in $z$ will be considered later in section 7, we ignore them for now. Then the potential $\phi$ is given by the following exponential expansion

$$\phi = 2\pi u_x u_y \sum_{(p,q) \neq (b,\bar{b})} \frac{e^{u_p z} + e^{u_p (\lambda z - z)}}{u_p (e^{u_p z} - 1)} e^{i u_q x} e^{i u_p y} + \ldots$$

Although this particular expansion is only valid for $z > 0$, analogous expansions of $\phi$ for $z < 0$ or along other directions for $x > 0$, $x < 0$, $y > 0$ and $y < 0$ can be obtained by simple coordinate transforms. The number of terms $\#\phi$ required in the expansion depending on the tolerance $\tau$ is given by (4.18), thus we have $\#\phi \rightarrow \infty$ as $z \rightarrow 0$. Consequently, this expansion should only be used for $z$ not too small. The potential $\phi_{b\bar{b}}$ may then be computed efficiently using the exponential expansion, as long as the distance between boxes $b$ and $\bar{b}$ is not too small. The implied split into far interactions and near interactions for a black box $b$ is shown graphically below.
The near contribution may be split further into a non-periodic Coulomb interaction $\phi_{1/r}$ from boxes $b$ close to box $b$ and a smooth interaction $\phi$ from the mirror images of boxes $b$.

The simple Coulomb potential $\phi_{1/r}$ may be computed pairwise. In contrast to this, the potential $\tilde{\phi}$ is given by the scary expression (4.14). Things are not so bad, however. Since $\tilde{\phi}$ is smooth for small $x$, $y$ and $z$, we may approximate it by an exponential expansion

$$\tilde{\phi} \approx \sum_{i,j,k} c_{ijk} \cosh(\alpha_i x) \cosh(\beta_j y) \cosh(\gamma_k z),$$

(6.2)

where we have used the fact that $\tilde{\phi}$ is even in $x$, $y$ and $z$. We will describe in section 7 how to determine the parameters $c_{ijk}$, $\alpha_i$, $\beta_j$ and $\gamma_k$ for the above approximation.

The above splits reduce the problem of computing $\phi$ to the following subproblems. For distant boxes $b$ and $\bar{b}$, we compute $\phi$ with the exponential expansion (6.1). For near boxes $b$ and $\bar{b}$ we have $\phi = \phi + \phi_{1/r}$, which may be computed with the exponential expansion (6.2) for $\tilde{\phi}$ and pairwise for $\phi_{1/r}$. Since we have described the pairwise problem in section 3, we are left with computing $\phi$ and $\tilde{\phi}$ by exponential expansions. Although the exponential expansions (6.1) and (6.2) are not identical, the structure to compute them efficiently is the same. Therefore, we will only consider the exponential expansion for $\phi$ in the following.

The problem may be reduced further by considering only the terms associated to indices $(p, q) + (p, -q) + (-p, q) + (-p, -q)$ in the expansion (6.1). We obtain the following expression

$$\phi_{pq}(x, y, z) = (\alpha_{pq}e^{u_{pq}z} + \delta_{pq}e^{-u_{pq}z}) \cos(u_q y) \cos(u_p x)$$

(6.3)

for some constant factors $\alpha_{pq}$ and $\delta_{pq}$. The potential $\phi$ may then be computed by combining the appropriate $\phi_{pq}$. To explain the algorithm, we will not use the three-dimensional $\phi_{pq}$, but a simpler two-dimensional $\psi_{pq}$ given by
\[ \psi_x(x, y) = (e^{-ux} + e^{ux}) \cos uy \quad \text{for} \quad x > 0. \]

The description of the problem in two dimension makes the formulas smaller and graphical representations easier. However, we have to make sure that essential properties of \( \phi \) are carried over to the two-dimensional function \( \psi_x \). Since \( \phi \) is symmetric with respect to permutation of coordinates \( x, y, \) and \( z \), we assume that we have to compute either \( \psi_x(x, y) \) or

\[ \psi_y(x, y) = (e^{-uy} + e^{uy}) \cos ux \quad \text{for} \quad y > 0. \]

Furthermore, we assume that the amount of work required to compute \( \psi_x \) is \( \#\psi = \lambda_x / x \), and the amount of work for \( \psi_y \) is \( \#\psi = \lambda_y / y \). Finally, we pretend that both \( \psi_x \) and \( \psi_y \) are periodic in \( x \) and \( y \) with periods \( \lambda_x \) and \( \lambda_y \), respectively. In the following subsections, we will write \( \psi \) to denote either \( \psi_x \) or \( \psi_y \), the distinction being clear from the context.

### 6.3 Product decomposition

Exponential and trigonometric functions have simple product decompositions due to the addition theorems. Given pairs of coordinates \((x, y)\) and \((\bar{x}, \bar{y})\) with \( y > \bar{y} \), we may write

\[ \psi(x - \bar{x}, y - \bar{y}) = e^{-uxy} \cos ux \times e^{uy} \cos u\bar{x} \]

\[ + e^{-uxy} \sin ux \times e^{uy} \sin u\bar{x} \]

\[ + e^{uy} \cos ux \times e^{-uxy} \cos u\bar{x} \]

\[ + e^{uy} \sin ux \times e^{-uxy} \sin u\bar{x}. \]

Furthermore, the partial derivatives of \( \psi(x - \bar{x}, y - \bar{y}) \) may be expressed using the same terms, which is another advantage of exponential expansions. The derivative with respect to \( x \) at \((x - \bar{x}, y - \bar{y})\) is

\[ \frac{\partial \psi}{\partial x} = e^{-uxy} \sin ux \times e^{uy} \sin u\bar{x} \times (-uy) \]

\[ + e^{-uxy} \cos ux \times e^{uy} \cos u\bar{x} \times (ux) \]

\[ + e^{uy} \sin ux \times e^{-uxy} \sin u\bar{x} \times (-uy) \]

\[ + e^{uy} \cos ux \times e^{-uxy} \cos u\bar{x} \times (ux). \]

and with respect to \( y \)

\[ \frac{\partial \psi}{\partial y} = e^{-uxy} \cos ux \times e^{uy} \cos u\bar{x} \times (-uy) \]

\[ + e^{-uxy} \sin ux \times e^{uy} \sin u\bar{x} \times (-uy) \]

\[ + e^{uy} \cos ux \times e^{-uxy} \cos u\bar{x} \times (ux) \]

\[ + e^{uy} \sin ux \times e^{-uxy} \sin u\bar{x} \times (ux). \]

For the true potential \( \phi \), the derivatives are essentially the forces \( f \) which we are interested in, maybe even more so than in the scalar potential \( \phi \). The main reason for writing these expressions down explicitly is to demonstrate the simplicity of the problem. Compute all possible
products of exponentials in $y$ and trigonometric functions in $x$, and combine them with analogous terms computed for $(\bar{x}, \bar{y})$. In two dimensions, the number of terms is four, while in three dimensions we have eight terms. In the following, we write the product decomposition formally as

$$\psi(x - \bar{x}, y - \bar{y}) = \sum_{a=1}^{4} \chi_a(x, y) \bar{\chi}_a(\bar{x}, \bar{y})$$

with obvious definitions of $\chi_a$ and $\bar{\chi}_a$.

### 6.4 Combining boxes

Given a formula $\psi(x, y)$ assumed to be valid for $y > 0$, we may compute the interaction potential for a particle at $(x, y)$ from another particle at $(\bar{x}, \bar{y})$ by $\psi(x - \bar{x}, y - \bar{y})$ given that $y > \bar{y}$. We will see later how to compute $\psi(x, y)$ for $y < \bar{y}$ by exploiting periodicity. Graphically, the situation with the periodic box outlined might look like

To compute the interaction potential $\psi(x, y)$ from several particles $(x_i, y_i)$, we may use the product decomposition and write

$$\sum_i \psi(x - x_i, y - y_i) = \sum_{i,a} \chi_a(x, y) \bar{\chi}_a(x_i, y_i) = \sum_a \chi_a(x, y) \sum_i \bar{\chi}_a(x_i, y_i), \quad (6.4)$$

which says that $\bar{\chi}$ terms for different particles $(x_i, y_i)$ may be combined by addition. The product decomposition of (6.4) can then be used to compute $\psi(x - \bar{x}, y - \bar{y})$ for a subset of particles $(\bar{x}, \bar{y})$ as shown below.

Equation (6.4) shows that the combined terms $\sum \bar{\chi}$ for $(x_i, y_i)$ particles are independent of $(x, y)$ given that $y > \bar{y}_i$. These combined terms may therefore be used for subsets of $(x, y)$ particles, as illustrated by the following picture.
Furthermore, instead of computing the interaction $\psi$ for pairs of boxes, we may combine the terms of several gray boxes and compute the interaction $\psi$ once for each black box of $(x, y)$ particles as shown below.

So far we are restricted by the condition $y > \tilde{y}$, but now we show how to compute the interaction $\psi$ for $y < \tilde{y}$. Due to the periodicity of $\psi$ we have for $y > 0$

$$\psi(x, -y) = \psi(x, \lambda y - y).$$

Therefore, we obtain for $y < \tilde{y}$

$$\psi(x - \tilde{x}, y - \tilde{y}) = \psi(x - \tilde{x}, y - (\tilde{y} - \lambda y)), \quad (6.5)$$

with $y > \tilde{y} - \lambda y$, which is the interaction potential $\psi$ of the particle $(\tilde{x}, \tilde{y})$ shifted by $\lambda y$ along the $y$ axis. Because of the periodic boundary conditions, this can be viewed as using a mirror image of the $(x, y)$ particle. Equation (6.5) corresponds to a simple shift by $\lambda y$, which implies that exponential factors $e^{\eta y}$ are multiplied by a factor $e^{-\eta \lambda y}$. Since this factor is independent of $\tilde{y}$, the same technique works equally well on the box level, where we have to shift sums of $e^{\eta y}$.

Note however that boxes with $y \approx \tilde{y}$ cannot be accounted for in the same way. Even if we could exclude the exceptional case $y = \tilde{y}$, the cost involved for $y \approx \tilde{y}$ is too high, since we have assumed $\#\psi = \lambda_x / (y - \tilde{y})$. Omitting gray boxes with $y \approx \tilde{y}$, we may therefore compute the interaction $\psi = \psi_y$ from the following region.

Since we may compute either $\psi_x$ or $\psi_y$, we may combine boxes in the same way as before, but along the $x$ axis, and we end up with the following situation for all possible $(x, y)$. 
The method described above shifts boxes only if mirror images are used. Another option is to write the exponential expansion in terms of local, box centered coordinates $\bar{y}$. In our two-dimensional example with 6 columns of boxes along the $y$ direction, this would imply shifts of $(-3\lambda_y/6, -2\lambda_y/6, -4\lambda_y/6)$ instead of $(0, 0, -\lambda_y)$ for the three gray columns. Since these shifts occur on the box level, there is little additional cost involved, but we gain simplicity due to identical shifts for each black box.

### 6.5 Mirror images

Let us determine the amount of work required to compute $\psi$ for all $n$ particles. To keep things simple, we assume that $\lambda_x = \lambda_y = 1$ and that we split the box into $b \times b$ small boxes, that is $b$ small boxes along the $x$ and $y$ directions each. Using the above scheme to combine boxes with $|y - \bar{y}| \geq 1/b$, we have $\#\psi \simeq b$ per particle. The average number of particles in each box is $n/b^2$, and if we assume that the pairwise near interaction $\psi$ costs $\#\psi = 1$ per pair, we obtain an approximation for the cost $c_\psi$ to compute all $\psi$ in two dimensions

$$c_\psi \simeq n(2b + 9n/b^2).$$

Since we are only interested in orders of $n$, we ignore the constant factors. Minimizing $c_\psi$ with respect to $b$, we obtain the following expressions in orders of $n$ for the optimal $b_\psi$ and $c_\psi$

$$b_\psi \sim n^{1/3} \quad \text{and} \quad c_\psi \sim n^{4/3}.$$

An analogous and equally crude approximation for the three-dimensional case $\phi$ goes along the same lines. We assume that $\lambda_x = \lambda_y = \lambda_z = 1$ and a splitting into $b \times b \times b$ small boxes, so that we have approximately $\#\phi \simeq b^2$ and an average number of $n/b^3$ particles per box. An estimate of the cost $c_\phi$ is then given by

$$c_\phi \simeq n(3b^2 + 27n/b^2).$$

Minimizing $c_\phi$ with respect to $b$, the following expressions in orders of $n$ drop out

$$b_\phi \sim n^{1/5} \quad \text{and} \quad c_\phi \sim n^{7/5}.$$

While $c_\phi \sim n^{7/5}$ is a significant improvement over $c \sim n^2$ for a pairwise computation, we can do even better than this by using a hierarchical box splitting.

For a given black box, we would like to combine gray boxes in a hierarchical way, using different levels of coarseness for combining gray boxes. The graphic below depicts this idea for two hierarchical levels.
We cannot do this with the current $\psi$ because we have lost periodicity after the first level of
the hierarchy. However, by adding appropriate mirror images, periodicity can be maintained
and the method can be applied for several levels. The graphic below illustrates the addition
of artificial mirror charges to transform a non-periodic $\lambda_x \times \lambda_y / 2$ box into a periodic box,
assuming the larger $\lambda_x \times \lambda_y$ box is periodic.

Assume that we want to compute $\psi$ for some particle at $(x, y)$, and for some reason a later
stage of the algorithm expects a particle at $(x', y')$. We may write

$$
\psi = \psi - \psi(x - x', y - y') + \psi(x - x', y - y'),
$$

where the term $\psi(x - x', y - y')$ represents the artificial particle, and the negative $-\psi(x -
x', y - y')$ is the corresponding compensation. If we can do the compensation on the box level,
the extra cost to maintain the periodicity is small, but it improves the time complexity of MMM
from $O(n^{7/5})$ to $O(n \log n)$. Graphically, the procedure of combining far contributions and
the compensation for mirror charges looks

So far, the creation of artificial mirror charges requires some extra shifts and additions on the
box level. We will see later that the computation of $\tilde{\phi}$ for particles $(\tilde{x}, \tilde{y})$ near $(x, y)$ is simpler
if the corresponding mirror charges are not inserted, which requires some additional work on
the box level.

We have shown the far contribution and the compensation for the $y$ direction in the first hier-
archy level, but it is straightforward to alternate directions and descend further levels. In the
graphic below, this process is indicated with true interactions shown gray and compensations
hatched.
Note that the number $b$ of boxes along the coordinate axis in the graphics is $b = 6$, which is the minimum required to compensate on the box level. In principle we could choose among $b = 6, 9, 12, \ldots$, but we will see below that $b = 6$ is the simplest and usually the most efficient choice. This restriction for $b$ on all levels except the last is implied by the $3 \times 3$ box structure near the black box and the requirement of compensation. When splitting along the $y$ axis, for example, the new $X'_y$ must be some integral fraction of $\lambda_y$. For $b = 6$ we have $X'_y = \lambda_y/2$, and for $b = 9$ we have $X'_y = \lambda_y/3$, for example.

To estimate the amount of work required depending on $b$, let us assume that $X_x = X_y = 1$ for the initial box, and that $b$ is the same for the $y$ and $x$ directions. The volume of the remaining region is reduced by a factor $b/3$ in each step. Therefore, we require $s_b = \log N / \log(b/3)$ steps to reduce the volume by a given factor $N$. The cost for a reduction along the $y$ direction is $c_y \simeq b$, the cost along the $x$ direction is $c_x \simeq b/(b/3) = 3$. A rough estimate of the overall cost $c_\psi$ in two dimensions given the above assumptions is

$$c_\psi \simeq (b + 3)s_b/2 \sim \frac{b + 3}{\log(b/3)},$$

which is minimal for $b_\psi \simeq 10.8$. Consequently, in the two-dimensional case $b = 9$ or $b = 12$ might be reasonable choices, assuming that any additional cost by introducing more mirror images can be neglected. An analogous analysis of the three-dimensional case starts with assumptions $\lambda_x = \lambda_y = \lambda_z = 1$ and a partitioning into $b \times b \times b$ small boxes. The formula for $s_b$ is the same, whereas the costs for reductions are given by $c_z \simeq b^2$, $c_y \simeq 3b$ and $c_x \simeq 9$. Then, a rough estimate of the cost in three dimensions is given by

$$c_\psi \simeq (b^2 + 3b + 9)s_b/3 \sim \frac{b^2 + 3b + 9}{\log(b/3)},$$

which is minimal for $b_\psi \simeq 6.0$. Consequently, in the three-dimensional case $b = 6$ seems the most reasonable choice, because it is the most efficient and simplest option. Note that this conclusion is based on assumptions which do not necessarily cover the general case. For instance, instead of $\lambda_1 = 1$, we might have $\lambda_x = 1$, $\lambda_y = \gamma$ and $\lambda_z = \gamma^2$ such that $X'_z = \lambda_z/(b/3) = \gamma^3$. In other words, the quotient of consecutive $\lambda$ is always $\gamma = (b/3)^{-1/3}$. The cost involved in one step is then

$$c_x = c_y = c_z \simeq \frac{\gamma \lambda_z \gamma^2 \lambda_z}{(\lambda_z/b)^2} = 3b.$$

Taking this variant into account, we obtain

$$c_\psi \simeq 3bs_b \sim \frac{b}{\log(b/3)},$$

which is minimal for $b_\psi \simeq 8.2$, which is fairly close to 9. While we should not trust our formula too much, inserting values $b = 6$ and $b = 9$ we obtain reasonable guesses of $c_\psi \simeq 8.7$ for $b = 6$ and $c_\psi \simeq 8.2$ for $b = 9$. Despite its slight disadvantage in this special case, we will stick to the case $b = 6$, which has a few other advantages, and which is at worst only slightly less efficient than $b = 9$. 

Mirror images
6.6 Combining levels

We have seen above how to compute all far interactions efficiently by introducing artificial mirror images. The problem with this approach is that these mirror images make the function \( \hat{\phi} \) less smooth, since the contributions move closer to the black box while descending levels, as illustrated below.

However, we may remove these unwanted contributions of artificial mirror images by adding correction terms on the box level for a 3 \( \times \) 3 region around the black box containing \((x, y)\). The graphic below shows the idea for two levels of refinement.

In section 7 we will describe how this can be implemented without too much overhead.

The following observation allows an efficient computation of terms associated to different hierarchy levels. Since we restrict ourselves to \( b_\phi = b_\beta = 6 \), the length of the periodic box is halved along a direction in each step. Therefore, if we start with a periodic box of size \( \lambda_x \times \lambda_y \), the size of the periodic box at the next hierarchy level is \( \lambda'_x \times \lambda'_y \) with \( \lambda'_x = \lambda_x / 2 \) and \( \lambda'_y = \lambda_y / 2 \). Correspondingly we have \( u'_x = 2u_x \) and \( u'_y = 2u_y \). Therefore, terms like \( e^{u'_x u'_y} \) may be computed as

\[ e^{u'_x u'_y} = e^{2u_x u_y} = (e^{u_x u_y})^2 \]

and similarly for trigonometric functions as

\[ \cos u'_x x = (\cos u_x x)^2 - (\sin u_x x)^2 \]
\[ \sin u'_x x = 2 \sin u_x x \cos u_x x. \]

Looking at our example function \( \psi(x, y) \) or at the series for \( \phi(x, y, z) \), we see that we may compute terms of each next hierarchy level by squaring certain expressions. Consequently, the implementation will compute the interaction terms for a sequence of levels along the same direction at once, as sketched below for three hierarchy levels.
The next subsection describes the overall algorithm to compute $\phi$ for a sequence of levels along the $z$ direction.

6.7 Computing $\phi$

Consider the term $\phi_{pq}$ given in (6.3), which is part of the sum for $\phi$. Let $\alpha = \alpha_{pq}$, $\bar{\alpha} = \bar{\alpha}_{pq}$ and $u = u_{pq}$ to simplify the notation. Then we have

$$\phi_{pq} = (\alpha e^{u \pi} + \bar{\alpha} e^{-u \pi}) \cos u_{pq} \cos u_{px}.$$

The term for the interaction $\phi_{pq}$ for particles $(x, y, z)$ and $(\bar{x}, \bar{y}, \bar{z})$ with charges $q$ and $\bar{q}$ is then given in product decomposed form by

$$\phi_{pq} = \alpha \cdot q e^{u \pi} \cos u_{pq} \cos u_{px} \times \bar{q} e^{-u \pi} \cos u_{pq} \cos u_{px}$$

$$+ \alpha \cdot q e^{u \pi} \sin u_{pq} \cos u_{px} \times \bar{q} e^{-u \pi} \sin u_{pq} \cos u_{px}$$

$$+ \alpha \cdot q \cdot q e^{u \pi} \sin u_{pq} \cos u_{px} \times \bar{q} e^{-u \pi} \sin u_{pq} \cos u_{px}$$

$$+ \alpha \cdot q \cdot q e^{u \pi} \sin u_{pq} \sin u_{px} \times \bar{q} e^{-u \pi} \sin u_{pq} \sin u_{px}$$

$$+ \bar{\alpha} \cdot q e^{-u \pi} \cos u_{pq} \cos u_{px} \times q e^{u \pi} \cos u_{pq} \cos u_{px}$$

$$+ \bar{\alpha} \cdot q e^{-u \pi} \cos u_{pq} \sin u_{px} \times q e^{u \pi} \sin u_{pq} \sin u_{px}$$

$$+ \bar{\alpha} \cdot q e^{-u \pi} \sin u_{pq} \cos u_{px} \times q e^{u \pi} \sin u_{pq} \sin u_{px}$$

$$+ \bar{\alpha} \cdot q e^{-u \pi} \sin u_{pq} \sin u_{px} \times q e^{u \pi} \sin u_{pq} \sin u_{px}.$$

Of course, there are some obvious simplifications if $p = 0$ or $q = 0$. This formula immediately suggests the following scheme for computing the far contribution of the interaction $\phi$. To avoid unnecessary complications, let us assume that we have two levels of refinement.

(i) For all particles $(x, \bar{y}, \bar{z})$ with charge $\bar{q}$, compute all combinations of

$$\bar{s}_{pq} = \bar{q} e^{u \pi} \cos u_{pq} \cos u_{px}$$

for all pairs $(p, q)$ required for the desired accuracy. The token cosin acts as a placeholder for either $\cos$ or $\sin$, just like $e^{u \pi}$ stands for $e^{u \pi}$ or $e^{-u \pi}$. The tuple $\bar{s}_{pq}$ contains 8 elements for $p \neq 0$ and $q \neq 0$, and 4 elements for $p = 0$ or $q = 0$. We may incorporate pairs $(p', q')$ of the second level into this notation by letting $p = 2p'$ and $q = 2q'$.

(ii) For all smallest boxes $\bar{b}$, compute the sums over all particles within $\bar{b}$ for all the terms computed in (i), symbolically

$$\bar{\sigma}_{pq} = \sum_{(x, \bar{y}, \bar{z}) \in \bar{b}} \bar{s}_{pq} = \sum_{(x, \bar{y}, \bar{z}) \in \bar{b}} \bar{q} e^{u \pi} \cos u_{pq} \cos u_{px}.$$

$$\cos u_{pq} \cos u_{px}$$

$$\cos u_{pq} \sin u_{px}$$

$$\sin u_{pq} \cos u_{px}$$

$$\sin u_{pq} \sin u_{px}.$$
Note that we may not combine yet boxes into larger regions, although some sketches seem to suggest this for outer hierarchy levels. These illustrations have ignored that we have to cancel artificial mirror images for near boxes to obtain a smooth φ. Since these cancellations take place on a smallest box basis, we need the terms for the smallest boxes on all levels.

(iii) For all smallest boxes b, compute all contributions of boxes \( \overline{b} \) corresponding to the appropriate level and the given direction. As one can see from the product decomposition above, this contribution takes the form of weighted terms

\[
\tilde{\Sigma}_{pq} = \sum_b \tilde{w}_{bg} \tilde{\sigma}_{pq} = \sum_b \left( \cdots \right) e^{\pm u_{pq} z} \cos u_q y \cos u_p x.
\]

(iv) For all particles \((x, y, z)\) with charge \(q\), compute the potential \(\phi\) and, if desired, the associated forces \(\partial \phi / \partial x, \ldots, \partial \phi / \partial z\) by using the formula of the product decomposition. The only difference is that we do not combine \((x, y, z)\) with a single \((\bar{x}, \bar{y}, \bar{z})\), but with a linear combination of \((\bar{x}, \bar{y}, \bar{z})\) particles. Symbolically, we may write

\[
\phi_{pq} = s_{pq} \times \tilde{\Sigma}_{pq}.
\]

Note that the terms required for \(s_{pq}\) are identical to the ones computed in (i), namely

\[
s_{pq} = q e^{\pm u_{pq} z} \cos u_q y \cos u_p x.
\]

In the next subsection, we briefly analyze a few variants of algorithms which rely on the scheme (i)–(iv) described above.

6.8 Choice of algorithm

In this subsection we discuss three substantially distinct algorithms to compute \(\phi\) for all levels along the z direction. Note that our comparison is mainly qualitative and concentrates on the aspects of minimizing overall memory requirements and the cost of memory accesses, taking into account the memory hierarchy.

A direct implementation of the description given above takes the following form

```latex
function \phi_1 \equiv
for all small boxes and \((p, q)\) do
for all particles do
   \(\tilde{s}_{pq} = \cdots\) -- need to store \(\tilde{s}_{pq}\)
od
\(\tilde{\sigma}_{pq} = \sum \tilde{s}_{pq}\)
od
for all small boxes and \((p, q)\) do
   \(\tilde{\Sigma}_{pq} = \sum \tilde{w} \tilde{\sigma}_{pq}\)
od
```
forall small boxes and \((p, q)\) do
forall particles do
\[ \phi_{pq} = \phi_{pq} + s_{pq} \times \Sigma_{pq} \quad \text{--- } s_{pq} \text{ already computed} \]
od

Note that there are still some open options in the ordering of the particles and \((p, q)\) loops. We will see later that computing \(s_{pq}\) for several \((p, q)\) at once allows some simplifications in \(\phi_1\) compared to \(\phi_2\) below. The main problem with the algorithm \(\phi_1\) is the amount of memory required for \(s_{pq}\) for all particles and all \((p, q)\). Let us assume that we need \#\(\phi\) indices \((p, q)\) in the sum for \(\phi\). Then the memory requirements for \(s_{pq}\) exceed those for all charges and coordinates \((q, x, y, z)\) and potentials and forces \((\phi, f_x, f_y, f_z)\) by about a factor \#\(\phi\). Since \#\(\phi\) \approx 50 even for a relatively modest accuracy in the approximation, the amount of memory required will be often considered too large.

The obvious remedy to this problem is to compute all \((p, q)\) terms independently as shown in the algorithm \(\phi_2\) below

function \(\phi_2 = \)
forall \((p, q)\) do
forall particles do
\[ \bar{s}_{pq} = \cdots \quad \text{--- need to store } \bar{s}_{pq} \]
od
forall small boxes do
\[ \bar{\Sigma}_{pq} = \sum s_{pq} \]
od
forall small boxes do
\[ \bar{\Sigma}_{pq} = \sum \bar{\phi} \bar{s}_{pq} \]
od
forall particles do
\[ \phi_{pq} = \phi_{pq} + s_{pq} \times \bar{\Sigma}_{pq} \quad \text{--- } s_{pq} \text{ already computed} \]
od

The additional amount of memory required by the algorithm \(\phi_2\) is about the same as the memory required for coordinates and forces, which is reasonable. However, the problem with \(\phi_2\) is that we may no longer exploit simplifications which are possible if \(s_{pq}\) for several \((p, q)\) are computed at once. Probably even worse, for each \((p, q)\) we have memory accesses to all coordinates and all forces, which will hurt performance unless the number \(n\) of particles is small. In this case, however, the amount of memory required by \(\phi_1\) might not be such a big problem anyway, so \(\phi_2\) may not be competitive even for small \(n\).

The third algorithm \(\phi_3\) tries to combine the advantages of \(\phi_1\) and \(\phi_2\) in the hope that some unavoidable disadvantages do not hurt performance considerably. We want to avoid storing all values \(s_{pq}\) while still computing values for several or all \((p, q)\) indices at once. The simple solution is to recompute \(s_{pq}\) in the combining phase instead of storing \(\bar{s}_{pq}\) earlier and reusing it. Execution time measurements in later section will reveal the additional cost of this recomputation.
function $\phi_3$
  forall small boxes do
    forall $(p, q)$ do
      forall particles do
        $s_{pq} = \cdots$ — no need to store $s_{pq}$
        $\sigma_{pq} = \sigma_{pq} + s_{pq}$
      od
    od
  forall $(p, q)$ do
    forall small boxes do
      $\Sigma_{pq} = \sum w_{pq}$
    od
  forall small boxes do
    forall $(p, q)$ do
      forall particles do
        $s_{pq} = \cdots$ — recomputation necessary
        $\phi_{pq} = \phi_{pq} + s_{pq} \times \Sigma_{pq}$
      od
    od
  od

The memory requirements of this algorithm are still substantial, so for lots of $(p, q)$ terms required it might make sense to split the $(p, q)$ region into a few chunks and perform an outer iteration on these chunks, just as we did in $\phi_2$ for single $(p, q)$ indices. In comparison to $\phi_1$, we need only $1/n_b$ times the amount of memory in $\phi_3$, where $n_b$ is the average number of particles per small box. In typical cases we will have $n_b \approx 20$, so the amount of memory required for $\phi_3$ is comparable to the memory required for coordinates and forces, as long as the required accuracy is not too high. Note the interchange of the loops in the middle phase, where we work separately on $(p, q)$ indices. In the next section, we will analyze in detail the implementations of the three loops in algorithm $\phi_3$. Symbolically, the three phases compute the following items

$$
\tilde{q} \rightarrow \tilde{\sigma} \rightarrow \tilde{\Sigma} \rightarrow \phi,
$$

where phases $q \rightarrow \sigma$ and $\tilde{\Sigma} \rightarrow \phi$ are primarily compute bound and $\tilde{\sigma} \rightarrow \tilde{\Sigma}$ is primarily memory bound. We will use the appropriate measures for the analysis.
7 MMM Implementation

This section describes the implementation of the MMM algorithm from section 6. We start with
the description of a few simple routines, which will be used as building blocks in the algorithms.
Then we analyze the computation of the far range potential \( \phi \), including the polynomial terms
in \( z \). We devote a separate subsection to the phase \( \hat{\sigma} \rightarrow \hat{\Sigma} \), where we introduce a method to
analyze the efficiency of memory bound routines in general. We conclude this section with a
description of the near range problem \( \tilde{\phi} \).

7.1 Basic routines

In this subsection we describe a few elementary routines which we considered worth optimiz¬
ing. The need for these routines will be seen later when we build the tables for the exponential
expansions. For each routine, we show the transition from the sequential code to the optimized
code using issue maps. For simplicity, we will often assume that some vector size \( \tilde{n} \) or \( \tilde{p} \) is
even or divisible by four, while the true implementation only assumes that \( \tilde{n} \) is even. This
more general problem may be solved by treating initial elements of the vector separately and
applying the optimized code to the rest of the vector. It may be a little tedious to write this
extra code, but it is not particularly difficult nor instructive, so we do not describe it in detail.
To demonstrate the importance of optimizing even simple problems, we measure the execution
times for one particular routine.

7.1.1 cseq — sequence of cosines and sines

Given a sequence of pairs \((\cos x_n, \sin x_n)\) for \( n = 1 \ldots \tilde{n} \), we want to compute the pairs
\((\cos px_n, \sin px_n)\) for all \( n \) and \( p = 1 \ldots \tilde{p} \). This can be done efficiently by using the addition
theorems

\[
\begin{align*}
\cos(p + 1)x &= \cos px \cos x - \sin px \sin x \\
\sin(p + 1)x &= \sin px \cos x + \cos px \sin x
\end{align*}
\]

to compute the sequences along the \( p \) direction. Assuming that \( \tilde{n} \) is even, the inner loop of our
code may compute \((\cos px_n, \sin px_n)\) pairs for \( p = 1 \ldots \tilde{p} \) and \( n = 1 \ldots 2 \). The issue maps of
these inner loops look as follows

Unlike in most other cases of software pipelining, a separate post-loop part is not necessary,
since the loop may be iterated \( \tilde{p} \) times instead of the expected \( \tilde{p} - 1 \). Note that some instructions
are executed unnecessarily during the last iteration of the loop, which is acceptable if there is
no possible side effect. In our case, there are no additional loads or stores, but the optimized
code as shown above is not equivalent to the sequential version with respect to floating point
exceptions. A quick comparison of the execution times required per pair \((\cos px_n, \sin px_n)\) for
the typical case \( \tilde{n} = 20 \) and \( \tilde{p} = 10 \) is given in the table below.
7.1.2 csqr — squaring cosines and sines

Given a sequence of pairs \((\cos x_n, \sin x_n)\) for \(n = 1 \ldots \tilde{n}\), we want to compute the pairs \((\cos 2x_n, \sin 2x_n)\) for all \(n\). This can be done using special cases of the addition theorems

\[
\cos 2x = (\cos x)^2 - (\sin x)^2 = (\cos x + \sin x)(\cos x - \sin x)
\]

\[
\sin 2x = 2\sin x \cos x = \sin x \cos x + \sin x \cos x.
\]

Note that there is some freedom in how to compute cosines and sines. To compute \(\cos 2x\), we may use \((2 \phi, 1 \phi)\) or \((1 \phi, 2 \phi)\), and computing \(\sin x\) requires either \((2 \phi)\) or \((1 \phi, 1 \phi)\). We assume that \(\tilde{n}\) is divisible by four, so we may compute the squares of four pairs simultaneously. This corresponds to fourfold loop unrolling except that we use \(\cos 2x = (\cos x)^2 - (\sin x)^2\) for two pairs and \(\cos 2x = (\cos x + \sin x)(\cos x - \sin x)\) for the other ones. The issue maps for the sequential and the optimized code look as follows.

![Issue Map for csqr](image)

7.1.3 csad — adding cosines and sines

Given two sequences \((\cos x_n, \sin x_n)\) and \((\cos y_n, \sin y_n)\) for \(n = 1 \ldots \tilde{n}\), we want to compute \((\cos(x_n + y_n), \sin(x_n + y_n))\) for all \(n\). This can be done efficiently by using the addition theorems

\[
\cos(x + y) = \cos x \cos y - \sin x \sin y
\]

\[
\sin(x + y) = \sin x \cos y + \cos x \sin y.
\]

We see that the number of floating point instructions is \(4 \phi\) and \(2 \phi\) for each pair \((c, s)\), so \(2 \phi\) slots remain unused. Since these two additions per \((c, s)\) are essentially for free, we compute the sums \(\sum_n \cos z_n\) and \(\sum_n \sin z_n\) for \(z_n = x_n + y_n\), which are needed by some applications. The issue maps for the sequential and the optimized code look as follows.

![Issue Map for csad](image)
7.1.4 prod — pairwise products
Given sequences \( (x_n) \) and \( (y_p) \) for \( n = 1 \ldots \tilde{n} \) and \( p = 1 \ldots \tilde{p} \), we want to compute \( z_{np} = x_n y_p \) for all \( n \) and \( p \). The basic idea of the implementation is to take small chunks from the \( n- \) and \( p\)-sequences and compute the pairwise products between these chunks. In our implementation, the chunk sizes are 4 and 2 for the \( n- \) and \( p\)-sequences, respectively, and the inner loop iterates along \( n \). For simplicity, we assume that \( \tilde{n} \) is divisible by 4 and that \( \tilde{p} \) is even. The issue maps for the sequential code of the inner loop and the optimized code look as follows.

7.1.5 pows — sequence of powers
Given a sequence \( (x_n) \) for \( n = 1 \ldots \tilde{n} \), we want to compute \( x_p^p \) for all \( p = 1 \ldots \tilde{p} \). The idea of an efficient implementation is to take a few \( x_n \) values and iterate along \( p \) in the inner loop. The chunk size is chosen to be 4, and we assume that \( \tilde{n} \) is divisible by 4 and that \( \tilde{p} \) is even. Then the issue maps of the inner loop look.

It is again possible to avoid an extra post-loop part by executing the loop \( \tilde{p}/2 \) times instead of \( \tilde{p}/2 - 1 \). As mentioned earlier, this may lead to floating point exceptions which would not occur in the original code due to unnecessary floating point multiplications.

7.2 Computing \( \phi \)
This subsection describes in detail the implementation of the algorithm \( \phi \), where we will use the notation \( \tilde{q} \rightarrow \tilde{\sigma} \), \( \tilde{\sigma} \rightarrow \tilde{\Sigma} \) and \( \tilde{\Sigma} \rightarrow \phi \) introduced earlier to denote the three phases of the algorithm. We describe each phase and briefly analyze the efficiency of important components. Finally, we show how to compute the terms 1, \( z \) and \( z^2 \) in the formula
\[
\phi = 2\pi u_x u_y \sum_{p,q} \cdots + 2\pi u_x u_y (u_z z^2 - z + \lambda_z / 6).
\]

7.2.1 Computing \( \tilde{q} \rightarrow \tilde{\sigma} \)
This subsection describes how to compute \( \tilde{\sigma}_{pq} \) for all \( (p,q) \) for a single small box. For illustration purposes we assume that the range of \( (p,q) \) required is given by \( \sqrt{p^2 + q^2} \leq 8.5 \). Furthermore we assume that we have two levels of refinement. Remember that \( (p', q') \) indices
from the second level translate into an index \((2p', 2q')\) of the first level, and we assume that the same restriction \(\sqrt{(p')^2 + (q')^2} \leq 8.5\) holds for the indices of the second level. The region in \((p, q)\) space where we have to compute \(\delta_{pq}\) then looks

In computing \(\delta_{pq}\) we do not care about the level the associated \((p, q)\) belongs to. To make the graphic less voluminous, and in view of the structure of the algorithm to be explained later, we introduce another notation which easily generalizes to more than two levels. With the above assumptions and an arbitrary number of levels, the region in \((p, q)\) space looks

This decomposition makes the regions for all levels look equal except for a small piece in the first level. This uniformity allows a simple loop structure in the algorithm. Our goal is to exploit addition theorems of trigonometric functions and simple product formulas of exponential functions to reduce the number of function evaluations when computing \(\delta_{pq}\). We have observed earlier that descending one level of the hierarchy corresponds to squaring corresponding values, which can be done for instance by using the optimized \texttt{csqr} routine. Given the values of the first level, we may therefore hop from one level to the next by simple and cheap operations. We are therefore left with the problem of computing \(e^{\pm u_{pq} z}\), \(\cos(u_{pq} y)\) and \(\cos(u_{pq} x)\) for one level of \((p, q)\) indices. All \(\cos(u_{pq} x)\) terms can be computed from the starting term \(\cos(u_{1} x)\) for \(p = 1\), and then using the \texttt{csqe} routine explained earlier for \(p = 1 \ldots p_{\text{max}}\).

The sequence \(\cos(u_{pq} y)\) can be computed analogously.

Note that the \(\cos{\text{in}}\) terms are relatively harmless because we only have to compute \(p_{\text{max}} + q_{\text{max}}\) of them, compared to about \(p_{\text{max}} q_{\text{max}}\) indices \((p, q)\). We are not equally lucky with the exponential terms, but we may nevertheless avoid some function evaluations. We know that \(u_{pq}\) is proportional to \(\sqrt{p^2 + \alpha q^2}\) for some constant \(\alpha\), in particular we have \(u_{pq} \sim p\) for \(q = 0\) and \(u_{pq} \sim q\) for \(p = 0\). Therefore, exponential terms for \(p = 0\) or \(q = 0\) may be computed as sequences of powers \(x, x^2, x^3, \ldots\), for instance by the routine \texttt{powx}. For nonzero \(p\) and \(q\) we may still use the scaling property of \(u_{pq}\), namely \(u_{kp, kq} = k u_{pq}\), so we may compute \(e^{\pm u_{kp, kq} z}\) with little work from \(e^{\pm u_{pq} z}\). Further simplifications are possible in the symmetric case \(\alpha = 1\),
but we will not try to exploit this. Considering these simplifications, we partition the \((p, q)\) region of the first level into two parts. One part contains the \((p, q)\) indices where we have to explicitly evaluate the exponential function, the other part contains exponentials which may be computed indirectly.

In our example, we need 35 function evaluations and 29 values may be computed indirectly. For the implementation it makes sense to sacrifice some performance to gain simplicity, so instead of exploiting arbitrary \(k\) in the relation \(u_{kp,kq} = ku_{pq}\), we restrict ourselves to \(k = 2\). Furthermore, we will compute all exponentials within the inner part of the \((p, q)\) region for nonzero \(p\) and \(q\) directly. This implies that we will not have to square exponentials more than once to compute all required values in the first level. Graphically, the regions of direct and indirect computation then look as follows.

In our example, this simplification results in 40 function evaluations and 24 indirect computations. Note that we have split the whole \((p, q)\) region of the first level into subregions I–VI which will be used later in the explanation of the algorithm. The simplifications mentioned above make sure that all exponentials within region III are computed directly.

Observe that regions I–III only occur at the first level, while regions IV–VI are repeated for further levels. Regions I,II and equally regions IV,V and their counterparts in deeper levels IV',V' require special treatment due to the special cases \(p = 0\) and \(q = 0\), respectively. The algorithm to compute \(\bar{\sigma}\) for all levels and a single small box may then be written as

\[
\text{function } \bar{q} \longrightarrow \bar{\sigma} \equiv \\
\text{forall particles do} \\
\text{eval } e^{\pm u_{pq} x} \\
\text{eval } \cos(u_{px}) \\
\text{eval } \cos(u_{qy}) \\
\text{od} \\
\text{forall } (p, q) \in I \cup II \cup III \text{ do} \\
\bar{\sigma}_{pq} = \sum \bar{\xi}_{pq} \\
\text{od}
\]

---

some \((p, q)\) indirectly using \texttt{prod} and \texttt{pows}  
\(- p = 1 \) directly, else using \texttt{cseq}  
\(- q = 1 \) directly, else using \texttt{cseq}  
four terms in \(\bar{\xi}_{pq}\) for I–II  
eight terms in \(\bar{\xi}_{pq}\) for III
forall levels do
  forall \((p, q) \in \text{IV} \cup \text{V} \cup \text{VI}\) do
    \(\bar{\sigma}_{pq} = \sum \bar{s}_{pq}\)
    — four terms in \(\bar{s}_{pq}\) for IV–V
    — eight terms in \(\bar{s}_{pq}\) for VI
  od
  forall particles do
    \(e^{\pm u_{pq}z} \rightarrow e^{\pm 2u_{pq}z}\)
    — in regions IV–VI
    \(\cos(u_{pq}x) \rightarrow \cos(2u_{pq}x)\)
    — using \(\cos\)
    \(\cos(u_{pq}y) \rightarrow \cos(2u_{pq}y)\)
    — using \(\cos\)
  od
od

The remaining subproblems turn out to be small enough so that we may consider manual optimizations. We pick the most expensive piece of this algorithm, the computation of

\[
\bar{\sigma}_{pq} = \sum \bar{s}_{pq}
\]

for nonzero \(p\) and \(q\), which happens to be the case for regions III,VI,VI',..., and we use this simple piece of code to illustrate the optimizations performed.

Since \(\bar{s}_{pq}\) and \(\bar{s}_{pq}\) are the same except for the sign in the exponential, we will rename \(\bar{s} \rightarrow s\), \(\bar{\sigma} \rightarrow \sigma\) for the rest of this subsection. Remember that each \(s_{pq}\) consists of eight terms originating from all combinations of

\[q e^{\pm u_{pq}z} \cos(u_{pq}y) \cos(u_{pq}x).\]

Assume that there are \(\bar{n}\) particles in this particular small box, then the algorithm to compute \(\sigma_{pq}\) for all \((p, q)\) with \(p \neq 0\) and \(q \neq 0\) in a given region is

\[
\text{function } \sigma_{\neq 0} \equiv \\
\text{forall } (p, q) \in \text{region do} \\
\sigma = \sigma_{pq} \\
forsome n = 1 \ldots \bar{n} do \\
q_n = \ldots \\
\langle e_+, e_- \rangle = e^{\pm u_{pq}z_n} \\
\langle c_p, s_p \rangle = \cos(u_{pq}x_n) \\
\langle c_q, s_q \rangle = \cos(u_{pq}y_n) \\
\sigma = \sigma + \langle q_n e_+ c_p c_q, q_n e_+ s_p s_q, q_n e_- c_p c_q, q_n e_- s_p s_q, q_n e_- s_p c_q, q_n e_- c_p s_q, q_n e_+ s_p c_q, q_n e_+ c_p s_q \rangle \\
e^{\pm u_{pq}z_n} \rightarrow e^{\pm 2u_{pq}z_n} \\
\sigma_{pq} = \sigma \\
\end{align*}
\]

— 8 loads
— routine adds previous values
— 8 stores
Note that there are no function evaluations at this stage, since the exponentials, cosines and sines have been computed earlier. Note further that the exponential values \( e^{\pm npg} \) for nonzero \( p \) and \( q \) are squared within the routine \( \sigma \neq 0 \), and not in the routine \( \sigma \) itself. To optimize this routine, we start with the dependence graph of its inner loop, as we have done several times by now. In the case of \( \sigma \neq 0 \), the structure of dependences and operations is fairly regular, as can be seen below.

Besides a few loop instructions not shown here, the inner loop contains 16 \( \bullet \), 8 \( \bullet \) and 9 \( \bullet \) elementary operations overall, so our goal is to have 1 \( \bullet \) each cycle. We can see from the dependence graph that there are relatively few dependences, so scheduling should not be particularly difficult. The issue maps of the sequential and the optimized code, which is obtained by loop unrolling twice and partial software pipelining, are shown below.

Because of the simple structure of the sequential code, we expect that a compiler is able to generate nearly optimal code for it. So let us look at the execution times for the routine \( \sigma \neq 0 \) on different processors, using \( n = 16 \) for the number of particles in the small box and \( m = 37 \) for the number of \((p, q)\) indices required. The choice \( m = 37 \) corresponds to our example used in the graphics, assuming that we want to compute region VI, VI', ..., but some other \( m \) would not make a big difference, since the inner loop is over \( \bar{n} = 1 \ldots \bar{n} \). In the table below, the execution time \( t \) measures the time per \( s_{pq} \), which corresponds to 16 multiplies.

<table>
<thead>
<tr>
<th>processor</th>
<th>( t ) [ns]</th>
<th>( m )</th>
<th>( n )</th>
<th>( s_{pq} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164</td>
<td>62</td>
<td>130</td>
<td>16</td>
<td>37</td>
</tr>
<tr>
<td>[500 MHz]</td>
<td>45</td>
<td>105</td>
<td>16</td>
<td>37</td>
</tr>
<tr>
<td>UltraSPARC-II</td>
<td>91</td>
<td>98</td>
<td>16</td>
<td>37</td>
</tr>
<tr>
<td>[336 MHz]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIPS R10000</td>
<td>42</td>
<td>45</td>
<td>16</td>
<td>37</td>
</tr>
<tr>
<td>[195 MHz]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The difference between the manually optimized code and the sequential code is about a factor 1.5 on the Alpha. If we consider that the simple routine \( o^0 \) contributes significantly to the overall cost, this improvement is probably worth the effort. On the MIPS, the sequential and optimized implementations are about equal, and on the UltraSPARC-II, the sequential version is faster.

### 7.2.2 Computing \( \bar{\sigma} \rightarrow \bar{\Sigma} \)

We have already mentioned that the phase \( \bar{\sigma} \rightarrow \bar{\Sigma} \) is mainly memory bound. Therefore, it would be inappropriate to check the quality of our implementation by counting instructions and comparing the execution time to the theoretical peak performance. A more reasonable approach is to compare our code to sample pieces of code which are essentially limited by the bandwidth between main memory and processor. This analysis and the method in general are the topics of subsection 7.3. In this subsection, we will only describe a few important details of the implementation. For the illustrations we will use a two-dimensional analog of the problem, and we assume that the periodic box is partitioned into 12 x 12 small boxes.

To start with, let us consider the interface of the \( \bar{\sigma} \rightarrow \bar{\Sigma} \) to the outside world. In terms of data dependences, we may simply write \( q \rightarrow \bar{\sigma} \rightarrow \bar{\Sigma} \rightarrow \phi \), but it may still make sense for \( q \rightarrow \bar{\sigma} \) to use a different layout of \( \bar{\sigma} \) in memory than \( \bar{\sigma} \rightarrow \bar{\Sigma} \), for instance. After the phase \( q \rightarrow \bar{\sigma} \), the layout of \( \bar{\sigma} \) in memory is assumed to look as follows.

![Diagram](path/to/diagram.png)

The transformation \( \bar{\sigma}_{pq} \rightarrow \bar{\Sigma}_{pq} \) can be done independently for each \((p, q)\) and each item within \( \bar{\sigma}_{pq} \), and symbolically we write \( \bar{\sigma}_i \rightarrow \bar{\Sigma}_i \). However, it is a bad idea to compute \( \bar{\Sigma}_i \) independently for each \( i \) on the above layout for the following reason. When loading values from main memory to a cache, this usually happens in chunks called cache lines. The width of cache lines depends on the specific hardware and maybe configuration, but we will assume for now that a cache line is 32 or 64 bytes wide, which corresponds to 4 or 8 double precision values. Given the above layout, loading some \( \bar{\sigma}_i \) means loading some of its neighbors \( \bar{\sigma}_{i+} \) as well, which are not used for computing \( \bar{\Sigma}_i \). Unless these neighbors are still in the cache when they are actually used, we are wasting bandwidth by transferring complete cache lines, but only using a small part of them. In principle, there are two ways to improve the situation. One possibility is to use most of the values within the same cache line, which in our case corresponds to using \( \bar{\sigma}_i \) and some of its neighbors at the same time. So, instead of combining \( \bar{\sigma}_i \rightarrow \bar{\Sigma}_i \) independently for each \( i \), we do it in small groups of 4 or 8 indices \( i \). The problem with this approach is that the amount of memory required is increased by the same factor 4 or 8, if we ignore cache lines. For a 12 x 12 x 12 grid and a cache line of 64 bytes, for instance, the amount of memory required is about 110K bytes. In the case of the Alpha 21164 with a second-level cache of 96K this is clearly too much. However, we do not need all of the 110K at the same time, and although it complicates some algorithms, it is possible to
order the computations in such a way that we need only about 50K and still fully exploit cache lines. A completely different approach is to change the data layout between phases $\bar{\phi} \rightarrow \bar{\sigma}$ and $\bar{\sigma} \rightarrow \bar{\Sigma}$ such that we have a simple layout to store $\bar{\sigma}$ and may nevertheless exploit full cache lines later. For this purpose, we combine the $\bar{\sigma}$ values of $2 \times 2 \times 2$ subregions such that $\bar{\sigma}$, belonging to these 8 small boxes are stored contiguously in memory. In our two-dimensional example, the combination of $2 \times 2$ boxes would look like

In three dimensions 8 values $\bar{\sigma}_i$ are stored contiguously in memory, and with the same assumptions as above the memory requirements reduce to about 14K, which is still too large for the first-level date cache of 8K on the Alpha 21164, but which fits comfortably into the 96K second-level cache. Consequently, our implementation shuffles the data along $\bar{\sigma} \rightarrow 8 \times \bar{\sigma}$ at the begin of phase $\bar{\sigma} \rightarrow \bar{\Sigma}$, and reshuffles it along $8 \times \bar{\Sigma} \rightarrow \bar{\Sigma}$ at its end.

The pattern for combining several $\bar{\sigma}_{pq}$ into $\bar{\Sigma}_{pq}$ depends on both the level the $(p,q)$ index belongs to and on the amount of reduction which has already happened along other coordinate axes. For the two-dimensional example and $12 \times 12$ small boxes we have the following 4 possible patterns, where the contributions due to mirror images are shown in light grey.

For the patterns on the coarser levels it makes sense to compute the regular contribution on a coarse grid, and only the cancellation of mirror images on the finest grid. In our example, the contributions of this coarse part may be computed by first combining the small boxes into $2 \times 2$ or $4 \times 2$ clusters, thereby reducing the amount of work required by a factor of 4 or 8, respectively. The only part which has to be done on the smallest box level is the cancellation of mirror images, as shown graphically below for one of the coarse patterns.

Note that we have indicated the coarse boxes appropriately. This technique may be applied for even coarser levels, as well. Another option for very coarse levels is not to cancel the mirror images, as long as these mirror images do not disturb the smoothness of the near range potential.
If the approximation of the potential \( \phi \) can cope with true mirror images occurring naturally in a \( 12 \times 12 \) partition, then it is sufficient to cancel mirror images on the two finest levels. For efficiency reasons, it is worthwhile to have separate routines for the cases

\[
H - J
\]

and

\[
m
\]

which correspond to 4 cases in three dimensions. To explain the ideas involved and describe the algorithm in detail, we choose the contribution on the finest level which combines 9 smallest boxes in the two-dimensional case. This process for all black boxes is shown in the illustration below, where we have switched to a combination along the \( y \) direction.

In three dimensions, this process corresponds to the combination of 27 gray boxes for each black box. Since there are no mirror images to consider, the illustrated problem is a little simpler than the other cases, but the general characteristics are more or less the same. A detailed analysis of this procedure and a description of methods to measure performance for memory bound routines is given in subsection 7.3.

### 7.2.3 Computing \( \bar{\Sigma} \rightarrow \phi \)

The structure of the algorithm \( \bar{\Sigma} \rightarrow \phi \) is similar to the computation of \( \bar{\phi} \rightarrow \bar{\sigma} \), since we are working on the same \( (p, q) \) regions and may therefore reuse the subdivision introduced for the phase \( \bar{\phi} \rightarrow \bar{\sigma} \). Furthermore, our decision to recompute \( s_{pq} \) for the computation of the potential \( \phi_{pq} = s_{pq} \times \bar{\Sigma}_{pq} \) makes the first part of the algorithm nearly identical to the computation of \( \bar{\sigma}_{pq} \). Making use of the procedures \( \text{prod}, \text{pows}, \text{csqf} \) and \( \text{csqr} \) as indicated in the algorithm \( \bar{\phi} \rightarrow \bar{\sigma} \), we have

```plaintext
function \( \bar{\Sigma} \rightarrow \phi \equiv 
for all \((p, q)\) and particles do
  eval \( e^{\pm \nu_{pq} z} \) — some \((p, q)\) indirectly
  eval \( \cos(\nu_{pq} x) \) — only \( p = 1 \) directly
  eval \( \cos(\nu_{pq} y) \) — only \( q = 1 \) directly
od
for all \((p, q) \in I \cup II \cup III\) and particles do
  \( \phi_{pq} = s_{pq} \times \bar{\Sigma}_{pq} \) — \( s_{pq} \) having 4 or 8 terms
  \( \phi = \phi + \phi_{pq} \)
od
```

forall levels do — IV–VI is IV'–VI', ... in deeper levels
  forall (p, q) ∈ IV ∪ V ∪ VI and particles do
    \( \sigma_{pq} = \sum \sigma_{pq} \) — \( \sigma_{pq} \) having 4 or 8 terms
  od
  forall (p, q) and particles do
    \( e^{\pm u_{pq} z} \rightarrow e^{\pm 2u_{pq} z} \) — in regions IV–VI
    \( \cos(p, x) \rightarrow \cos(2p, x) \)
    \( \cos(p, y) \rightarrow \cos(2p, y) \)
  od
od

Note that we have used \( \phi_{pq} \) to represent both the potential and the associated forces. Now we describe the computation of \( s_{pq} \times \Sigma_{pq} \) in more detail. As we have done in the case \( \vec{q} \rightarrow \vec{\sigma} \), we look at the computation for regions III or VI, that is for nonzero \( p \) and \( q \), since this is the most expensive piece of the overall algorithm \( \Sigma \rightarrow \phi \). Let \( \bar{n} \) be the number of particles within the small box we want to update \( \phi \) for, then the algorithm looks as follows

function \( \phi_{\neq 0} \equiv \)

forall (p, q) ∈ region do
  \( \Sigma_{(0...7)} = \Sigma_{pq} \)
  \( c(x, y, z) = \langle u_p, u_q, u_{pq} \rangle \) — factors for derivatives
  forall \( n = 1...\bar{n} \) do
    \( \langle e_+, e_- \rangle = e^{\pm u_{pq} z_n} \) — 2 loads
    \( \langle e_p, s_p \rangle = \cos(u_p, x_n) \) — 2 loads
    \( \langle e_q, s_q \rangle = \cos(u_q, y_n) \) — 2 loads
    \( s = \langle e_+, c_p c_q, e_+, c_p s_q, e_+, s_p c_q, e_+, s_p s_q \rangle \)
    \( \phi = \langle (s_4 \Sigma_0 + s_5 \Sigma_1 + s_6 \Sigma_2 + s_7 \Sigma_3 + s_0 \Sigma_4 + s_1 \Sigma_5 + s_2 \Sigma_6 + s_3 \Sigma_7)_x, \)
    \( (s_0 \Sigma_0 + s_1 \Sigma_1 - s_2 \Sigma_2 - s_3 \Sigma_3 + s_4 \Sigma_4 + s_5 \Sigma_5 - s_6 \Sigma_6 - s_7 \Sigma_7)_x \rangle \times c_x, \)
    \( (s_5 \Sigma_0 - s_4 \Sigma_1 + s_3 \Sigma_2 - s_2 \Sigma_3 + s_1 \Sigma_4 - s_0 \Sigma_5 + s_6 \Sigma_6 - s_7 \Sigma_7)_y \rangle \times c_y, \)
    \( (s_4 \Sigma_0 + s_5 \Sigma_1 + s_6 \Sigma_2 + s_7 \Sigma_3 - s_0 \Sigma_4 - s_1 \Sigma_5 - s_2 \Sigma_6 - s_3 \Sigma_7)_z \rangle \times c_z \)
  \( \phi_n = \phi_n + \phi \)
  \( e^{\pm u_{pq} z_n} \rightarrow e^{\pm 2u_{pq} z_n} \) — 4 loads and 4 stores
od

Note that the charge \( q_n \) of the particles is not used in this algorithm, since we may just as well multiply the final \( \phi_n \) by the charge \( q_n \) once. Furthermore, we perform the squaring of the exponentials here and not in the outer routine \( \Sigma \rightarrow \phi \), just as we did in the routine \( \Sigma_{\neq 0} \). Of course we exploit the occurrence of common subexpressions to minimize the number of operations required. We may compute \( s = \langle \cdots \rangle \) via subexpressions \( e_+, e_- \), \( c_p c_q \), \( c_p s_q \), \( s_p c_q \) and \( s_p s_q \) with a total of 12 multiplications instead of 16 for a naive implementation. Furthermore, the sums \( s_4 \Sigma_0 + s_5 \Sigma_1 + s_6 \Sigma_2 + s_7 \Sigma_3 \) and \( s_0 \Sigma_4 + s_1 \Sigma_5 + s_2 \Sigma_6 + s_3 \Sigma_7 \) may be used for both \( \phi \) and \( \phi_2 \). Considering this, we end up with the following dependence graph for the inner loop of the algorithm \( \phi_{\neq 0} \).
Note that we assumed that it is not necessary to shorten the dependence chain by restructuring
additions in $s_6 \Sigma_0 + s_7 \Sigma_1 - s_4 \Sigma_2 - s_5 \Sigma_3 + s_2 \Sigma_4 + s_3 \Sigma_5 - s_0 \Sigma_6 - s_1 \Sigma_7$, for instance. Adding
one term after the other has the disadvantage of reducing parallelism unnecessarily, but the
advantages are that it is simpler, requires less registers, and leads to more evenly distributed
multiplications and additions. To illustrate this last point, assume we compute the sum in a
binary tree-like fashion, which in our example is simply $((s_0 \Sigma_0 + s_7 \Sigma_1) - (s_4 \Sigma_2 + s_5 \Sigma_3)) +$
$((s_2 \Sigma_4 + s_3 \Sigma_5) - (s_0 \Sigma_6 + s_1 \Sigma_7))$. Then all 8 multiplications have to be done after at most 4
additions, with 3 more additions to come, while in the sequential version we require only 6
multiplications being done at this point. An instruction count shows that we have 41 $\times$, 26 $\circ$ and 16 $\bullet\bullet\bullet\bullet$ for each iteration of the inner loop. Since we have many floating point
operations, optimization is pretty straightforward, and the issue maps of the sequential code
and the software pipelined version are shown below.

The situation regarding the expected gain by manually optimizing the code is similar to the
case $\tilde{q} \rightarrow \tilde{\sigma}$. There are few loads and stores, lots of multiplications and additions, and a
relatively simple structure of dependences. Therefore, the compiler should be able to optimize
this piece of code fairly well. We measure the execution times on different processors for
$\tilde{n} = 16$ particles in the small box and $\tilde{m} = 37$ indices $(p, q)$ within the region of interest. Since
the inner loop is over $n = 1 \ldots \tilde{n}$, the particular value of $\tilde{m}$ is not so important. The execution
times in the table below measure the contribution per $s_{pq} \times \Sigma_{pq}$, which takes 41 multiplications
in our implementation.
Again, we have an improvement of about a factor 1.5 of the manually optimized version on the Alpha over the sequential version, which is optimized by the compiler. Somewhat surprisingly, the other processors perform worse in comparison to the Alpha for the \( \phi_{\neq 0} \) procedure compared to the \( \phi_{\neq 0} \) procedure, if we look at the execution time ratios. Note that the 41 multiplications imply an optimal execution time of 82 ns, so the 90 ns attained show that the clock rate and the amount of pipelining in the floating point unit are the limiting factors for this routine.

### 7.2.4 The terms 1, \( z \) and \( z^2 \)

We have ignored so far the contribution

\[
\varphi(z) = 2\pi u_x u_y (u_z z^2 - z + \lambda_z / 6) = c_0 1 + c_1 z + c_2 z^2
\]

to the potential \( \phi \). The expression \( \varphi(z - \bar{z}) \) has an obvious product decomposition

\[
\varphi(z - \bar{z}) = c_0 1 \times 1 + c_1 z \times 1 + c_2 z^2 \times 1 - c_1 1 \times \bar{z} - 2c_2 z \times \bar{z} + c_2 1 \times \bar{z}^2.
\]

The amount of work required for \( \varphi \) is small compared to the sum of exponential terms, so efficiency is not a problem. Our primary goal is to use the same routines in phase \( \sigma \rightarrow \bar{\Sigma} \) for both the exponential terms of \( \phi \) as well as the polynomial terms of \( \varphi \). Remember that \( \bar{\Sigma} \) is essentially a sum of shifted \( \sigma \), and shifting an exponential corresponds to a multiplication by some factor. We may therefore write

\[
\bar{\Sigma} = \sum_\sigma \alpha_\sigma \bar{\sigma}
\]

for some factors \( \alpha_\sigma \). The obstacle to using the same routines for the polynomial terms is that a shift \( z \rightarrow z + \zeta \) does not naturally correspond to a simple multiplication.

In the phases \( \bar{g} \rightarrow \bar{\sigma} \) and \( \bar{\Sigma} \rightarrow \phi \) this is not a problem, since we may work more or less directly on \( z \) or \( \bar{z} \) coordinates. For the former case, assume that

\[
\varphi = \chi_0 \times 1 + \chi_1 \times \bar{z} + \chi_2 \times \bar{z}^2,
\]

where \( \chi_* \) are functions in \( z \). Then a shift \( \bar{z} \rightarrow \bar{z}' = \bar{z} + \bar{\zeta} \) can be applied in a straightforward way onto the \( \bar{z} \) coordinates, namely

\[
1 \rightarrow 1, \quad \bar{z} \rightarrow \bar{z} + \bar{\zeta}, \quad \bar{z}^2 \rightarrow \bar{z}^2 + 2\bar{z}\bar{\zeta} + \bar{\zeta}^2.
\]
For the phase \( \Sigma \rightarrow \phi \) we assume \( \varphi \) being given in the form
\[
\varphi = 1 \times \bar{x}_0 + z \times \bar{x}_1 + z^2 \times \bar{x}_2,
\]
where \( \bar{x}_i \) are functions in \( \bar{z} \). A shift \( \bar{z} \rightarrow \bar{z} + \bar{\zeta} \) corresponds to a shift in the \( z \) coordinate \( z \rightarrow z - \zeta \), so we may just as well look at the case \( z \rightarrow z + \zeta \). Note that this shift occurs at a point when \( \bar{x}_i \) are already computed, but \( z \) is still unspecified. This means that we have to transform \( \bar{x}_i \) whereas above we could transform \( z \) directly
\[
\bar{x}_0 \rightarrow \bar{x}_0 + \zeta \bar{x}_1 + \zeta^2 \bar{x}_2, \quad \bar{x}_1 \rightarrow \bar{x}_1 + 2\zeta \bar{x}_2, \quad \bar{x}_2 \rightarrow \bar{x}_2.
\]
For the phase \( \phi \rightarrow \Sigma \) we have to express the shift \( \bar{z} \rightarrow \bar{z} + \zeta \) in terms of multiplications of components, which may then be combined later. After the shift, the potential \( \varphi \) may be written
\[
\varphi = \chi_0 \times 1 + \chi_1 \times \bar{z} + \chi_2 \times \bar{z}^2
+ \chi_1 \times \zeta + \chi_2 \times 2\zeta \bar{z} + \chi_2 \times \zeta^2.
\]
The terms in this product decomposition contain only products, so this form can be used for implementing a shift by multiplications. The price to be paid is that we have to extend the tuple \( (1, \bar{z}, \bar{z}^2) \). Formally, the shift may be written
\[
\langle 1, \bar{z}, \bar{z}^2, 1, \bar{z}, 1 \rangle \rightarrow \langle 1, \bar{z}, \bar{z}^2, \zeta, 2\zeta \bar{z}, \zeta^2 \rangle.
\]
Note that this method only works for at most one shift, which is fortunately the case for our combining routines in phase \( \phi \rightarrow \Sigma \). We may recapitulate this simple method by the following diagram
\[
\langle 1, \bar{z}, \bar{z}^2 \rangle \rightarrow \langle 1, \bar{z}, \bar{z}^2, 1, \bar{z}, 1 \rangle \quad \downarrow \quad \downarrow
\langle 1, \bar{z} + \zeta, \bar{z}^2 + 2\zeta \bar{z} + \zeta^2 \rangle \leftarrow \langle 1, \bar{z}, \bar{z}^2, \zeta, 2\zeta \bar{z}, \zeta^2 \rangle
\]
Since efficiency is not crucial for this small part of the computation, we can afford to increase the tuple size. In contrast to this, the code to combine \( \phi \rightarrow \Sigma \) is fairly voluminous, since different cases are implemented separately for efficiency reasons. So the amount of programming work saved by this simple idea to shift \( z \) and \( z^2 \) can hardly be overestimated.

### 7.3 Analysis of \( \phi \rightarrow \Sigma \)

In this subsection we describe in detail how to analyze the performance of a routine which is primarily memory bound. The particular example we have in mind is the computation of \( \phi \rightarrow \Sigma \) on the last level without mirror images. Graphically, the problem we want to solve is given by
We first introduce a simple method to find a reasonable upper bound of the expected performance for codes which are memory bound. Second, we describe the combining algorithm on the level of memory accesses and will draw some conclusions. Finally, a particular implementation of the algorithm is compared against the idealized version and shown to be nearly optimal.

### 7.3.1 Memory accesses

Obviously, the possible performance of some code is ultimately limited by the available hardware. For instance, the peak performance on the Alpha 21164 is $10^9$ floating point operations per second for a clock rate of 500 MHz. There are other limitations like the number and type of instructions issued per cycle and so forth. When we analyze some routine, we may compare the measured performance to an ideal performance based on characteristics like issues per cycle or floating point operations per cycle. This type of analysis makes sense if most memory accesses are within the primary or maybe secondary cache. If the algorithm accesses slower memory, neglecting the cost of memory accesses will not work any more. So it is unavoidable to consider the cost of accessing slow memory for this type of algorithm.

While it is possible to estimate the cost due to memory accesses depending on the given hardware for a particular code, this relies on a lot of detailed information and may nevertheless not be very accurate. And although this approach is undeniably useful in some cases, it may easily turn out to be too complicated. Another approach is to actually implement several variants of the algorithm and pick the best one, with the disadvantage of much more programming work. A reasonable compromise is not to implement the complete algorithm, but only the coarse pattern of memory accesses, and to ignore any data dependences. For illustration, assume we want to implement a fast daxpy routine, that is a routine which computes

$$y_n := y_n + \alpha x_n$$

for vectors $x_n$ and $y_n$ with $n = 1 \ldots \bar{n}$. Using the dependence graph, the transition from the algorithm to its pattern of memory accesses, $\mu$ pattern for short, looks like

The transition removes all operations except loads and stores, and the only remaining dependences are between loads and stores to the same address, so that the order in time remains the same. In the graphic, this time ordering constraint is indicated by a dashed line. This reduction to memory accesses may look trivial, but it is actually useful to determine to what extent an algorithm is limited by memory accesses alone.

Implementing this $\mu$ pattern is straightforward, since there are no restrictions except the time ordering of loads and stores. Note that this time ordering is not a data dependence, so we will
usually store the value 0 and not some value loaded earlier. For the analysis we assume that \( n \) is a multiple of 8. A simple algorithm implementing the \( \mu \) pattern of the daxpy routine is then given by

\[
\text{function daxpy}_\mu \\
\quad \text{for } n = 0, 8, \ldots, \bar{n} - 8 \text{ do} \\
\quad \quad \text{load } x_n \ldots x_{n+7} \\
\quad \quad \text{load } y_n \ldots y_{n+7} \\
\quad \quad \text{store } y_n \ldots y_{n+7} \\
\quad \od\]

Sometimes we might improve the performance by preloading values early, instead of simply relying on the loads and stores. In our case, preloading is not too important, since we do not have any data dependences, and the improvement by adding some preloads would be at most a few percent on the Alpha 21164. We will use preloads in some of our later experiments, however.

For the rest of this subsection, all the measurements are done on a particular machine containing an Alpha 21164, the technical details given in appendix C. Note that the processor itself is not really relevant in this context. The execution times of the daxpy and daxpy\(\mu\) routines for varying numbers \( \bar{n} \) of elements are given in the table below, with times in ns per element.

<table>
<thead>
<tr>
<th>( \bar{n} )</th>
<th>( \text{daxpy}_\mu )</th>
<th>daxpy</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^7 )</td>
<td>5.0</td>
<td>8.9</td>
</tr>
<tr>
<td>( 2^8 )</td>
<td>5.0</td>
<td>8.9</td>
</tr>
<tr>
<td>( 2^9 )</td>
<td>5.0</td>
<td>8.9</td>
</tr>
<tr>
<td>( 2^{10} )</td>
<td>5.0</td>
<td>10.0</td>
</tr>
<tr>
<td>( 2^{11} )</td>
<td>5.0</td>
<td>9.9</td>
</tr>
<tr>
<td>( 2^{12} )</td>
<td>5.0</td>
<td>9.9</td>
</tr>
<tr>
<td>( 2^{13} )</td>
<td>16.8</td>
<td>17.8</td>
</tr>
<tr>
<td>( 2^{14} )</td>
<td>31.8</td>
<td>31.8</td>
</tr>
<tr>
<td>( 2^{15} )</td>
<td>33.9</td>
<td>33.9</td>
</tr>
</tbody>
</table>

The conclusions we may draw from this particular analysis is that the daxpy implementation cannot be improved for large values of \( \bar{n} \), unless we find a more efficient memory access pattern. There is some room for improvement for smaller \( \bar{n} \), which corresponds to the case when the vectors \( x_n \) and \( y_n \) fit into the second-level cache of 96K for the Alpha 21164.

To get an impression on the available memory bandwidth on the machine we are performing these experiments right now, we may look at even simpler patterns. The first one simply loads values sequentially from memory, and the second one stores to the same memory locations in addition to the loads. The following execution times are measured for a varying number \( \bar{n} \) of elements, the times being given in ns per element.
Note that the entries \( \tilde{n} = 2^{20} \) and \( \tilde{n} = 2^{21} \) correspond to accessing 8M and 16M of main memory, so we see that the external cache is about 8M. The size of the second-level cache can be seen in the transitions \( \tilde{n} = 2^{13} \to 2^{14} \to 2^{15} \) corresponding to memory accesses of 64K \( \to 128K \to 256K \). The size of the external cache depends on the particular machine and not on the processor, so optimizations assuming a particular size of the external cache should be avoided. However, as the table above demonstrates, it is important to keep the memory requirements as modest as possible on any level, not just on first and second-level caches.

### 7.3.2 The \( \mu \) pattern for \( \tilde{\sigma} \to \tilde{\Sigma} \)

In this subsection, we examine the \( \mu \) pattern for an idealized phase \( \tilde{\sigma} \to \tilde{\Sigma} \) first, and then we gradually move towards a more realistic model. In the two-dimensional case, a simple and efficient method to compute \( \tilde{\Sigma} \) is based on combining values first along the \( x \) and then along the \( y \) direction. Graphically, this idea can be represented as:

\[
\begin{array}{c}
\tilde{\sigma} \\
\tilde{\Sigma}_x \\
\tilde{\Sigma}_{xy} = \tilde{\Sigma}
\end{array}
\]

Note that instead of \( \tilde{\sigma} \to \tilde{\Sigma}_x \to \tilde{\Sigma} \) we could use the other path \( \tilde{\sigma} \to \tilde{\Sigma}_y \to \tilde{\Sigma} \). For our problem the combination along \( x \) consists simply of additions, while the combination along \( y \) is more complicated for two reasons. First, we need some multiplications along \( y \), and second, the distance between the combined values is larger, so we will probably require more registers.

In the three-dimensional case, the situation is very similar. The combinations along \( x \) and \( y \) are simple additions, while the combination along \( z \) is more complicated. We may similarly write the ordering of combinations as \( \tilde{\sigma} \to \tilde{\Sigma}_x \to \tilde{\Sigma}_{xy} \to \tilde{\Sigma}_{xyz} = \tilde{\Sigma}, \) for instance. The combinations along \( x \) and \( y \) are identical, so if we ignore the data layout in memory, there are three essentially different orderings for combinations along \( x, y \) and \( z \). For the following we assume that the algorithm to compute \( \tilde{\Sigma}_{xyz} \) is given by:

```python
function eval_S_{xyz} :=
forall z-slices do
  eval \( \tilde{\Sigma}_x \) — single slice
  eval \( \tilde{\Sigma}_{xy} \) — single slice
od

eval \( \tilde{\Sigma}_{xyz} \) — all slices
```
The \( \mu \) pattern derived from this algorithm reduces the evaluations to pure load-store sequences, i.e., we assume that each value of the corresponding region is read from memory and written to memory once. Note that the ordering of the computations is such that \( \Sigma_{xy} \) reads values from a slice which was previously written to. For concreteness, let us assume that the periodic box is partitioned into 12 \( \times \) 12 \( \times \) 12 small boxes. Then the memory required per slice is \( \leq 1.2 \text{K} \) and for all small boxes \( \leq 14 \text{K} \). Therefore, a single slice will comfortably fit into the first-level cache on the Alpha 21164, while all slices together fit into the second-level cache. For a partitioning into 24 \( \times \) 24 \( \times \) 24 small boxes, a single slice still fits into the 8K first-level cache, but the second-level cache is too small to contain the data of all boxes. This difference alone makes a common analysis of the 12 \( \times \) 12 \( \times \) 12 and 24 \( \times \) 24 \( \times \) 24 cases too cumbersome. We will therefore consider the partitioning fixed and handle only the 12 \( \times \) 12 \( \times \) 12 case in the quantitative analysis.

The computations for \( \Sigma_{xyz}, \Sigma_{zxy} \) and \( \Sigma_{yxz} \) are different because the combination along \( z \) differs from the combinations along \( x \) or \( y \). In the \( \mu \) pattern, however, all combinations correspond to load-store sequences, so we have to look only at a single \( \Sigma_{\mu} \). We obtain more realistic results if we take into account that the data for a single \( \sigma_i \rightarrow \Sigma_i \) is not laid out sequentially in memory, but consists of octets which are spread in memory. The graphic below shows the comparison of the idealized and the realistic memory layouts. The times are given in ns per box and per item \( \sigma_i \), and we have assumed a total of \( n = 31, 63, 127 \) or 255 items. The number of items to combine determines the gap between octets and therefore the number of such octets within the same page. A large gap implies that we will have to reload page table entries more often. The symbol \( \Sigma_{\mu\sigma} \) denotes the \( \mu \) pattern with a sequential data layout, \( \Sigma_{\mu} \) the \( \mu \) pattern with octets, \( \Sigma_{xyz} \) the real computation with ordering \( x \rightarrow y \rightarrow z \) and \( \Sigma_{zxy} \) analogously for the ordering \( z \rightarrow x \rightarrow y \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>31</th>
<th>63</th>
<th>127</th>
<th>255</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Sigma_{\mu\sigma} )</td>
<td>28</td>
<td>28</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>( \Sigma_{\mu} )</td>
<td>36</td>
<td>38</td>
<td>41</td>
<td>42</td>
</tr>
<tr>
<td>( \Sigma_{xyz} )</td>
<td>41</td>
<td>45</td>
<td>47</td>
<td>48</td>
</tr>
<tr>
<td>( \Sigma_{zxy} )</td>
<td>43</td>
<td>47</td>
<td>51</td>
<td>51</td>
</tr>
</tbody>
</table>

Note that we have assumed for the illustration that the cost increases monotonically with the number \( n \) of items to compute. The conclusions we may draw from these timings are the following. First, the non-sequential layout contributes significantly to the cost of this routine, as we can see by comparing \( \Sigma_{\mu\sigma} \) and \( \Sigma_{\mu} \). Second, the difference between the \( \mu \) pattern alone and the real computation is only about 15\%, more or less independent of \( n \), as one can see by comparing \( \Sigma_{\mu} \) and \( \Sigma_{xyz} \). Third, the ordering is not very important, although \( \Sigma_{xyz} \) seems to be a little better than \( \Sigma_{zxy} \), but this could be the result of yet another detail in the data layout, which we will not discuss here. The important point is that \( \Sigma_{xyz} \) is a very simple routine, so we can be fairly confident that its implementation is close to optimal, and may therefore be regarded as an upper limit in performance for the real computation.

We see that \( \mu \) pattern analysis can be a very valuable tool for problems like \( \sigma \rightarrow \Sigma \). Once we have achieved a performance close to the optimum predicted by the simple \( \mu \) pattern, any further substantial improvement requires a modification of the \( \mu \) pattern. And experiments to optimize the \( \mu \) pattern itself are a lot simpler than working on the true algorithm due to the simple structure of the memory access code.
7.4 Computing $\tilde{\phi}$

In this subsection we describe how to compute the smooth part $\tilde{\phi}$ of the Coulomb interaction $\phi$ for particles and boxes which are close to each other. We have seen in section 6 that this near contribution splits into a non-periodic Coulomb interaction $\phi_{1/r}$ and a smooth potential $\tilde{\phi} = \phi - \phi_{1/r}$ given by expression (4.14). Graphically, the problem we want to solve is the following

Note that we have to compute this type of problem for each small black box. Due to the introduction of artificial mirror charges, these black boxes do not necessarily belong to the same large periodic box. However, this introduces no additional complication, since the formula is independent of the location of the small box within the periodic box.

7.4.1 Approximation

Let us assume that $A_\epsilon = 1$ and that we have $12 \times 12 \times 12$ small boxes. Then we have to compute $\phi$ for $|x|, |y|, |z| \leq 1/6$. Since $\tilde{\phi}$ is an even function in $x, y$ and $z$, a simple approximation of $\tilde{\phi}$ is given by products of cosh as follows

$$\tilde{\phi} \approx \sum_{i,j,k} c_{ijk} \cosh \alpha_i x \cosh \alpha_j y \cosh \alpha_k z,$$

for some coefficients $c_{ijk}$ and $\alpha_s$. Note that the coefficients $\alpha_s$ are the same for coordinates $x, y$ and $z$ due to symmetry for $A_\epsilon = 1$. In more general cases, if $\lambda_x \neq \lambda_y$ for instance, we will have different coefficients $\alpha_i, \beta_j$ and $\gamma_k$ for the coordinates $x, y$ and $z$. In the symmetric case we expect the coefficients $c_{ijk}$ to be symmetric with respect to permutations of their indices, so $c_{123} = c_{213}$ for instance. An alternative to cosh terms are cos terms, so we will compare the two expansions below.

Given the exponents $\alpha_s$, the coefficients $c_{ijk}$ may be determined such that the maximal absolute error $\epsilon = \epsilon_{\text{max}}$ is minimized. It is reasonable to let $\alpha_0 = 0$ since then we have $\cosh \alpha_0 x = 1$, which reduces the work required for terms containing the coefficient $c_0$. For the cubic $12 \times 12 \times 12$ problem, we obtain the following maximal errors $\epsilon$ depending on $n_{\alpha}$, the number of $\alpha_s \neq 0$. The table displays the error $\epsilon$ of the cosh and cos expansions as $-\log_{10} \epsilon$, that is the number of correct decimal digits.

<table>
<thead>
<tr>
<th>$n_{\alpha}$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>cosh</td>
<td>4.4</td>
<td>5.8</td>
<td>7.4</td>
<td>9.7</td>
</tr>
<tr>
<td>cos</td>
<td>3.1</td>
<td>6.0</td>
<td>7.4</td>
<td>10.0</td>
</tr>
</tbody>
</table>
Note that $\tilde{\phi} \approx -2.7$, so the relative error $\epsilon_{rel}$ is approximately $\epsilon_{rel} \approx 0.4 \times \epsilon$. The amount of computation required for cos terms is larger than for cosh terms due to more complicated combinations and force evaluations. The results indicate that the cosh expansion is more accurate than the cos expansion for $n_\alpha = 1$. For more terms, both methods are similarly accurate with a slight advantage for the cos expansion for high accuracies. The attainable accuracy depends not only on the number of terms in the expansion, but on the region we have to approximate $\tilde{\phi}$ for. The results above are valid for a $12 \times 12 \times 12$ partitioning of a cube, analogous results for a $24 \times 24 \times 24$ partitioning are given in the table below. The maximal error is again given as $-\log_{10} \epsilon$ and listed for a few $n_\alpha$.

<table>
<thead>
<tr>
<th>$n_\alpha$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>cosh</td>
<td>6.3</td>
<td>8.4</td>
<td>9.8</td>
<td>12.5</td>
</tr>
<tr>
<td>cos</td>
<td>4.3</td>
<td>7.9</td>
<td>9.8</td>
<td>12.6</td>
</tr>
</tbody>
</table>

The characteristics of the errors are pretty much the same as before. The cosh expansion is more accurate than the cos expansion for few terms, and is slightly less accurate for many terms. We may conclude therefore that the cosh expansion is only slightly less accurate than the cos expansion in the worst case. Considering that the cosh expansion requires less work for the same number of terms, our approximations will be of the cosh type from now on.

### 7.4.2 Implementation

The overall structure of the computation of $\tilde{\phi}$ using the cosh expansion is very similar to the computation of the far range $\phi$, which splits into phases $\tilde{\eta} \rightarrow \tilde{\sigma} \rightarrow \tilde{\Sigma} \rightarrow \tilde{\phi}$. For this reason we will use the same notation again with a last phase $\tilde{\Sigma} \rightarrow \tilde{\phi}$, in the hope that there is not too much confusion because of this reuse of symbols. Let the approximation for $\tilde{\phi}$ be given as

$$\tilde{\phi} = \sum_{i,j,k} c_{ijk} \cosh \alpha_i x \cosh \beta_j y \cosh \gamma_k z$$

with $\alpha_0 = \beta_0 = \gamma_0 = 0$. For a particle $(x, y, z)$ located in box $b$, we have to compute the interaction $\tilde{\phi}$ with all particles $(\tilde{x}, \tilde{y}, \tilde{z})$ in the same box $b$ or one of the neighbor boxes of $b$. These interactions $\tilde{\phi}$ may be computed for all particles $(x, y, z)$ using the following scheme

(i) For all particles $(\tilde{x}, \tilde{y}, \tilde{z})$ with charge $\tilde{q}$, compute all combinations of

$$s_{ijk} = \tilde{q} e^{\pm \alpha_i \tilde{x}} e^{\pm \beta_j \tilde{y}} e^{\pm \gamma_k \tilde{z}}$$

for all triples $(i, j, k)$ required for the desired accuracy.

(ii) For all small boxes, compute the sum of all $s$ of particles within this box

$$\bar{s}_{ijk} = \sum s_{ijk},$$

separately for all triples $(i, j, k)$. We will write this symbolically as $\bar{s} = \sum s$. 
(iii) For all small boxes, compute the contribution of the box itself and all neighbor boxes

\[ \bar{\Sigma}_{ijk} = \sum \bar{\omega} \bar{\sigma}_{ijk}. \]

Note that this is simply a linear combination of the \( \bar{\sigma} \) terms of the neighboring boxes. Symbolically, we abbreviate this combination by \( \bar{\Sigma} = \sum \bar{\sigma} \) and the complete phase as \( \bar{\Sigma} \rightarrow \bar{\phi}. \)

(iv) For all particles \((x, y, z)\) with charge \(q\), compute the potential \(\bar{\phi}\) and the associated forces \(\partial \bar{\phi}/\partial x, \partial \bar{\phi}/\partial y, \partial \bar{\phi}/\partial z\) by using the product decomposition of the cosh expansion described above. The formula for the product decomposition is written symbolically as \(s_{ijk} \times \bar{\Sigma}_{ijk}\) and this final phase as \(\bar{\Sigma} \rightarrow \bar{\phi}\).

We will not describe this process in detail except for the most important routine, namely the computation of \(\bar{\phi}\) for nonzero \(i, j\) and \(k\) in the phase \(\bar{\Sigma} \rightarrow \bar{\phi}\). It may be useful for the reader to compare this routine with the analogous routine \(\bar{\phi}_{\neq 0}\) used for the far contribution. Let \(\bar{n}\) be the number of particles within the small box \(b\). Then the algorithm to compute the near contribution is given by

\[
\text{function } \bar{\phi}_{\neq 0} =
\begin{align*}
\text{forall } (i, j, k) \neq 0 \text{ do} \\
\bar{\Sigma}_{(i\ldots j\ldots k)} = \bar{\Sigma}_{ijk} \\
\{x, y, z\} = \{\alpha_i, \beta_j, \gamma_k\} & \quad \text{--- factors for derivatives} \\
\text{for } n = 1 \ldots \bar{n} \text{ do} \\
\{x_+, x_-\} = e^{\pm \alpha_i} & \quad \text{--- 2 loads} \\
\{y_+, y_-\} = e^{\pm \beta_j} & \quad \text{--- 2 loads} \\
\{z_+, z_-\} = e^{\pm \gamma_k} & \quad \text{--- 2 loads} \\
& \quad s = \{x_+ + z_+, x_+ y_+ z_+, x_+ y_+ z_+, x_+ y_+ z_+\} \\
& \quad \phi = \{(s_7 \bar{\Sigma}_0 + s_6 \bar{\Sigma}_1 + s_5 \bar{\Sigma}_2 + s_4 \bar{\Sigma}_3 + s_3 \bar{\Sigma}_4 + s_2 \bar{\Sigma}_5 + s_1 \bar{\Sigma}_6 + s_0 \bar{\Sigma}_7), \\
& \quad (s_7 \bar{\Sigma}_0 + s_6 \bar{\Sigma}_1 + s_5 \bar{\Sigma}_2 + s_4 \bar{\Sigma}_3 + s_3 \bar{\Sigma}_4 - s_2 \bar{\Sigma}_5 - s_1 \bar{\Sigma}_6 - s_0 \bar{\Sigma}_7) \times c_x, \\
& \quad (s_7 \bar{\Sigma}_0 + s_6 \bar{\Sigma}_1 + s_5 \bar{\Sigma}_2 + s_4 \bar{\Sigma}_3 - s_2 \bar{\Sigma}_5 + s_1 \bar{\Sigma}_6 - s_0 \bar{\Sigma}_7) \times c_y, \\
& \quad (s_7 \bar{\Sigma}_0 - s_6 \bar{\Sigma}_1 + s_5 \bar{\Sigma}_2 - s_4 \bar{\Sigma}_3 + s_3 \bar{\Sigma}_4 + s_2 \bar{\Sigma}_5 - s_1 \bar{\Sigma}_6 - s_0 \bar{\Sigma}_7) \times c_z\} \\
& \quad \phi_n = \phi_n + \phi & \quad \text{--- 4 loads and 4 stores} \\
\text{od} \\
\text{od}
\end{align*}
\]

It can be seen from this algorithm why the cosh expansion is cheaper than the cos expansion for the same number of terms. With the cosh expansion we require only the term \(s_7 \bar{\Sigma}_0\) for both the potential and its derivatives, for instance. With the cos expansion, we would have to compute all of \(s_7 \bar{\Sigma}_0, s_6 \bar{\Sigma}_0, s_5 \bar{\Sigma}_0\) and \(s_4 \bar{\Sigma}_0\). A quick instruction count of the routine \(\bar{\phi}_{\neq 0}\) gives 23 \(\Theta\), at most 20 \(\Theta\) if we exploit common subexpressions, and 4 \(\Theta\) for each iteration of the inner loop. In the next subsection, we will analyze the efficiency of a particular implementation of \(\bar{\phi}_{\neq 0}\).

The computation for \(i, j\) and \(k\) all being nonzero is different from other cases only because we have chosen \(\alpha_0 = \beta_0 = \gamma_0 = 0\), which significantly reduces the amount of work, as we
now show. Assume that \( n_\alpha \) is the number of nonzero \( \alpha, \beta, \) and \( \gamma \). Then we have \( n_\alpha^3 \) triples \((i, j, k)\) with no zero index, \( 3n_\alpha^2 \) triples with exactly one zero index, \( 3n_\alpha \) triples with exactly two zero indices, and 1 triple \((0, 0, 0)\). The number of scalar elements within \( \bar{\sigma} \) for each of these cases is 8, 4, 2 and 1 respectively. So the total number of elements is given by

\[
\#\bar{\sigma} = 8n_\alpha^3 + 12n_\alpha^2 + 6n_\alpha + 1.
\]

If \( n_\alpha = 3 \) for example, the number of scalar elements to be computed is \( \#\bar{\sigma}_{\neq 0} = 216 \) for the nonzero triples \((i, j, k)\) and \( \#\bar{\sigma} = 343 \) overall. If we would not treat the cases \( \alpha_0 = 0, \beta_0 = 0 \) or \( \gamma_0 = 0 \) separately, the number of elements would be

\[
\#\bar{\sigma} = 8(n_\alpha + 1)^3 = 8n_\alpha^3 + 24n_\alpha^2 + 24n_\alpha + 8,
\]

and therefore \( \#\bar{\sigma} = 512 \) overall for \( n_\alpha = 3 \).

### 7.4.3 Efficiency

The analysis of the phases \( \bar{q} \longrightarrow \bar{\sigma} \longrightarrow \bar{\Gamma} \longrightarrow \bar{\varphi} \) for the near contribution is similar to the earlier analysis for the far contribution. Therefore, we restrict ourselves to the analysis of the routine \( \bar{\varphi}_{\neq 0} \) in this subsection. We skip the dependence graph of \( \bar{\varphi}_{\neq 0} \) and jump directly to the issue map and the illustration of the optimizations involved.

Since the number of registers does not suffice for a straightforward implementation, it was necessary to reload some \( \bar{\Gamma} \) values in each iteration. These load instructions are shown in dark grey in the above issue map.

The execution times of this routine for different processors and compile options are given below. The number of particles in the box is chosen \( n = 16 \) and the number of nonzero \( \alpha \) in the \( \cosh \) expansion used is \( n_\alpha = 3 \). Therefore, the number of \((i, j, k)\) tuples in the \( \bar{\varphi}_{\neq 0} \) routines is 27, but since the inner loop is over the \( n \) particles, a smaller \( n_\alpha \) would not change the results by much.

<table>
<thead>
<tr>
<th>processor</th>
<th>( t ) [us]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164</td>
<td>57</td>
</tr>
<tr>
<td>[500 MHz]</td>
<td>78</td>
</tr>
<tr>
<td></td>
<td>127</td>
</tr>
</tbody>
</table>
Since the minimal execution time is 48 $n s$ due to $23 \odot$ and $1 \odot$, the measured 57 $n s$ are within 20% of the optimum despite some non-negligible overhead outside the loop. On the Alpha, the optimized routine is more than twice as fast compared to the sequential code. Note that this improvement was possible in this case by simply software pipelining the inner loop carefully.
8 MMM Efficiency

In this section, we analyze the efficiency of the MMM implementation described in section 7. First, we measure the fraction of execution time spent in particular phases and subphases for a typical problem. After this profile analysis, we compare the efficiency of MMM to our implementation of PPPM. We have tried to make this comparison as fair as possible by using manually optimized routines for the short range part of PPPM. Then, we measure the true time complexity of MMM depending on the problem size n. We will see that the execution time of MMM is nearly linear in n for a range of $2^{10} \leq n \leq 2^{20}$ particles. Finally, we compare MMM to our implementation of the Ewald sum for small problems.

8.1 Profile

In this subsection we collect the profiling data of MMM for a typical problem. The problem is given by $n = 2^{15} = 32768$ particles in a $1 \times 1 \times 1$ periodic cube, with charges and coordinates randomly chosen. For a partition into $12 \times 12 \times 12$ small boxes and a tolerance $\tau = 10^{-3}$, we obtain the following fractions $t_f$ of the overall execution time $t$ for different phases of MMM.

<table>
<thead>
<tr>
<th>phase</th>
<th>$t_f [%]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>far contribution</td>
<td></td>
</tr>
<tr>
<td>$\bar{q} \rightarrow \bar{\sigma}$</td>
<td>14.7</td>
</tr>
<tr>
<td>$\bar{\sigma} \rightarrow \bar{\Sigma}$</td>
<td>11.0</td>
</tr>
<tr>
<td>$\bar{\Sigma} \rightarrow \phi$</td>
<td>23.8</td>
</tr>
<tr>
<td>other</td>
<td>6.3</td>
</tr>
<tr>
<td>near contribution</td>
<td></td>
</tr>
<tr>
<td>$\bar{q} \rightarrow \bar{\sigma}$</td>
<td>5.3</td>
</tr>
<tr>
<td>$\bar{\sigma} \rightarrow \bar{\Sigma}$</td>
<td>2.7</td>
</tr>
<tr>
<td>$\bar{\Sigma} \rightarrow \phi$</td>
<td>6.4</td>
</tr>
<tr>
<td>other</td>
<td>1.1</td>
</tr>
<tr>
<td>pairwise $1/r$</td>
<td></td>
</tr>
<tr>
<td>$q \rightarrow r^2$</td>
<td>5.4</td>
</tr>
<tr>
<td>$r^2 \rightarrow 1/r$</td>
<td>9.8</td>
</tr>
<tr>
<td>$1/r \rightarrow \phi_{1/r}$</td>
<td>9.7</td>
</tr>
<tr>
<td>other</td>
<td>1.2</td>
</tr>
<tr>
<td>other</td>
<td>2.6</td>
</tr>
</tbody>
</table>

Note that the tolerance $\tau$ imposes a limit on the absolute error of the pairwise potential $\phi$. In practice, this error estimate is too conservative, at least for uniformly distributed particles. Therefore, when comparing errors, it may be more appropriate to use the true error, and we will do so in the next subsection. The above profile shows that the most expensive part of MMM is the far contribution to $\phi$, so we will look closer at its most costly subphase $\bar{\Sigma} \rightarrow \phi$ in the table below. The times are again given in percent relative to the complete computation, only the scale of the graphical display has changed.
Obviously, the most prominent part of this computation is $\phi_{\neq 0}$ which we have analyzed in detail earlier. Now we analyze the cost of the far contribution for different values of the tolerance $\tau$.

In the table below, execution times in s are measured for the same typical problem, that is for $n = 2^{15}$ and $12 \times 12 \times 12$ small boxes.

| phase          | $t_f [\%]$ | | = phase, | = subphase |
|----------------|-------------|-----------------------------|-------------|
| far $\Sigma \rightarrow \phi$ | 23.8 | | | |
| $\phi_{\neq 0}$ | 12.1 | | | |
| $\phi_{p=0, q=0}$ | 3.0 | | | |
| $\phi_{p=q=0}$ | 0.5 | | | |
| $c_{\Sigma}$ | 3.0 | | | |
| $\cos x$ | 0.6 | | | |
| $c_{\text{seq}}$ | 0.8 | | | |
| $\text{pow}$ | 0.6 | | | |
| $\text{prod}$ | 0.5 | | | |
| $\text{csqr}$ | 0.3 | | | |
| other | 2.4 | | | |

| phase          | $t [s]$ | | = phase, | = subphase |
|----------------|---------|-----------------------------|-------------|
| far $\tau = 10^{-2}$ | 0.70 | | | |
| $\bar{q} \rightarrow \bar{\sigma}$ | 0.19 | | | |
| $\bar{\sigma} \rightarrow \Sigma$ | 0.15 | | | |
| $\Sigma \rightarrow \phi$ | 0.28 | | | |
| other | 0.08 | | | |
| far $\tau = 10^{-4}$ | 2.10 | | | |
| $\bar{q} \rightarrow \bar{\sigma}$ | 0.49 | | | |
| $\bar{\sigma} \rightarrow \Sigma$ | 0.43 | | | |
| $\Sigma \rightarrow \phi$ | 0.89 | | | |
| other | 0.29 | | | |
| far $\tau = 10^{-6}$ | 5.00 | | | |
| $\bar{q} \rightarrow \bar{\sigma}$ | 1.18 | | | |
| $\bar{\sigma} \rightarrow \Sigma$ | 1.15 | | | |
| $\Sigma \rightarrow \phi$ | 1.91 | | | |
| other | 0.76 | | | |
| far $\tau = 10^{-8}$ | 9.27 | | | |
| $\bar{q} \rightarrow \bar{\sigma}$ | 2.14 | | | |
| $\bar{\sigma} \rightarrow \Sigma$ | 2.31 | | | |
| $\Sigma \rightarrow \phi$ | 3.44 | | | |
| other | 1.38 | | | |

These results require some interpretation. Remember that the number of entries in the table for the far contribution is proportional to $\log^2 \tau$, and that the amount of work is more or less proportional to the number of entries in the table. If we decrease the tolerance from $\tau = 10^{-4}$ to $\tau = 10^{-6}$, we expect that the execution time increases by about a factor $9/4$. To analyze the above table we compare the actual increase of the execution time with the theoretical estimate.
\( \log^2 \tau \) by giving the corresponding ratios. Ratios < 1 indicate that the execution time increases less than predicted by the simple estimate, ratios > 1 indicate that the execution time increases more than expected for some transition like \( \tau = 10^{-4} \rightarrow 10^{-6} \). Note that the following table only lists these ratios, so there are no new measurements involved.

<table>
<thead>
<tr>
<th>Transition</th>
<th>( 10^{-2} \rightarrow 10^{-4} )</th>
<th>( 10^{-4} \rightarrow 10^{-6} )</th>
<th>( 10^{-6} \rightarrow 10^{-8} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Far contribution</td>
<td>0.75</td>
<td>1.06</td>
<td>1.04</td>
</tr>
<tr>
<td>( q \rightarrow \bar{\sigma} )</td>
<td>0.64</td>
<td>1.07</td>
<td>1.02</td>
</tr>
<tr>
<td>( \bar{\sigma} \rightarrow \bar{\Sigma} )</td>
<td>0.72</td>
<td>1.19</td>
<td>1.13</td>
</tr>
<tr>
<td>( \bar{\Sigma} \rightarrow \phi )</td>
<td>0.79</td>
<td>0.95</td>
<td>1.01</td>
</tr>
<tr>
<td>Other</td>
<td>0.90</td>
<td>1.16</td>
<td>1.02</td>
</tr>
</tbody>
</table>

The transition \( \tau = 10^{-2} \rightarrow 10^{-4} \) is seen to be special because the number of table entries for \( \tau = 10^{-2} \) is so small that a few optimized routines do not work particularly well. Furthermore, the simple approximation \( \log^2 \tau \) for the cost is not really appropriate for too large \( \tau \). For these reasons, it is not too surprising that the execution time does not increase by a factor 4 for the transition \( \tau = 10^{-2} \rightarrow 10^{-4} \). For the rest of the table the approximation works reasonably well, and there is a good explanation for the increased cost for the subphase \( \bar{\sigma} \rightarrow \bar{\Sigma} \) and other parts involving accesses to slow memory. Depending on the number of entries in the table, the collected values for \( \bar{\sigma} \) and \( \bar{\Sigma} \) fit or do not fit onto the off-chip cache. While the exact numbers are difficult to predict since we are working with tables of different sizes for each computation along a coordinate axis, the deviation of the measured data from the simple estimate \( \log^2 \tau \) is primarily a result of accessing slower memory. Despite this weakness, the execution times for the far contribution are more or less proportional to \( \log^2 \tau \) for \( \tau \) not too large or small, as the table above shows.

### 8.2 Comparison to PPPM

In this subsection we compare MMM to our implementation of the PPPM method described in section 5. The error \( \epsilon \) will be computed from the exact forces \( \mathbf{F}_i \) and the computed forces \( \mathbf{\tilde{F}}_i \) for all particles by

\[
\epsilon^2 = \frac{\sum_i \| \mathbf{F}_i - \mathbf{\tilde{F}}_i \|^2}{\sum_i \| \mathbf{F}_i \|^2}.
\]

Another simple measure of the error which is often used compares the potentials instead of the forces, namely

\[
\epsilon_\phi^2 = \frac{\sum_i (\phi_i - \phi_\tilde{i})^2}{\sum_i \phi_i^2}.
\]

For MMM we have roughly \( \epsilon_\phi \approx 2\epsilon \) in typical cases, so there is not a huge difference. The following table and illustration show the characteristics of the methods for a typical problem with \( n = 2^{15} = 32768 \) particles within a \( 1 \times 1 \times 1 \) periodic box. The coordinates and charges of the particles are uniformly distributed with the restriction of charge neutrality. It can be seen by experiment that the error \( \epsilon \) does not significantly change if the charges would be chosen \( q = \pm 1 \) instead. As usual, the execution times \( t \) are given in seconds.
### Time complexity

In this subsection we analyze the efficiency of MMM for varying problem size $n$, while keeping the error $\epsilon$ constant. The following experiments will be done for $\epsilon = 10^{-5}$. From the theory we expect the execution time to be roughly proportional to $n \log n$, where $n$ is the number of particles. Note that $O(n \log n)$ is the complexity of the far contribution, while the near

#### Table

<table>
<thead>
<tr>
<th>$\epsilon$</th>
<th>PPPM</th>
<th>MMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-2}$</td>
<td>0.3</td>
<td>1.0</td>
</tr>
<tr>
<td>$10^{-3}$</td>
<td>1.1</td>
<td>1.2</td>
</tr>
<tr>
<td>$5 \times 10^{-4}$</td>
<td>2.9</td>
<td>1.4</td>
</tr>
<tr>
<td>$10^{-4}$</td>
<td>*</td>
<td>1.7</td>
</tr>
<tr>
<td>$10^{-5}$</td>
<td>*</td>
<td>2.6</td>
</tr>
<tr>
<td>$10^{-6}$</td>
<td>*</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Note that the PPPM method is not applicable for arbitrary accuracies. For our particular implementation we obtain reasonable performance up to $\epsilon = 5 \times 10^{-4}$, with the cost growing quickly for higher accuracies. To determine the execution times for the PPPM method, it is necessary to find reasonable values for parameters like the discretization grid $m \times m \times m$, the parameter $\alpha$ and the cutoff radius $r$. The accuracy of the long range force is influenced by $m$ and $\alpha$, while the accuracy of the short range force is influenced by the parameters $\alpha$ and $r$. Assuming that the error $\epsilon$ is about evenly split between long range and short range force, the timings for errors $\epsilon_1 = 10^{-2}$, $\epsilon_2 = 10^{-3}$ and $\epsilon_3 = 5 \times 10^{-4}$ look roughly as follows, with times $t$ given in seconds:

#### Diagram

Note that while the precise results given above are only valid for our particular implementation of PPPM, it is generally true that PPPM is not able to solve problems with arbitrary accuracy efficiently. If the tolerated error $\epsilon$ is not too small, however, the PPPM method is more efficient than MMM. The data above indicates that this will be the case for errors of about $\epsilon \geq 10^{-3}$.
and pairwise contributions have only complexity $O(n)$. This is one reason the execution time might not increase as quickly as expected with increasing $n$. On the other hand, the amount of memory required grows linearly with $n$, which will make the execution slower than predicted by simple complexity considerations. Finally, the relation between the tolerance $\tau$ and the resulting error $\epsilon$ is not independent of the problem size. We might see that $\tau$ may actually increase with increasing $n$ and constant error $\epsilon$. Of course, the most reliable way to investigate these hypotheses is by experiment, and it turns out that the tolerance is about $\tau \approx 5 \times 10^{-4}$ to obtain a final error $\epsilon = 10^{-5}$. Depending on the number $n$ of particles, different partitions of the original $1 \times 1 \times 1$ box are useful. For the measurements below, the partitions are $\pi_6 = 12 \times 12 \times 12$, $\pi_7 = 12 \times 12 \times 24$, $\pi_8 = 12 \times 24 \times 24$, $\pi_9 = 24 \times 24 \times 24$ and $\pi_{10} = 24 \times 24 \times 48$. Note that the execution times $t$ will be given later in the graphic as $t \times \sigma/n$ for some scaling factor $\sigma$. The execution times $t$ in the table itself are given in seconds.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\pi_6$</th>
<th>$\pi_7$</th>
<th>$\pi_8$</th>
<th>$\pi_9$</th>
<th>$\pi_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{15}$</td>
<td>2.5</td>
<td>3.7</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>$2^{16}$</td>
<td>4.6</td>
<td>5.4</td>
<td>7.5</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>$2^{17}$</td>
<td>12.5</td>
<td>10.8</td>
<td>10.3</td>
<td>15.4</td>
<td>*</td>
</tr>
<tr>
<td>$2^{18}$</td>
<td>*</td>
<td>28.3</td>
<td>22.1</td>
<td>24.7</td>
<td>37.3</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>*</td>
<td>*</td>
<td>57.8</td>
<td>47.3</td>
<td>55.0</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>*</td>
<td>*</td>
<td>170.8</td>
<td>118.2</td>
<td>105.0</td>
</tr>
</tbody>
</table>

The memory requirements for the partition $24 \times 48 \times 48$ turn out to be too high for $\epsilon = 10^{-5}$ and the current implementation of MMM, so we cannot analyze problems for $n$ much larger than $2^{20}$. For the other end of the spectrum, when $n < 2^{15}$, we have to examine coarse partitions like $6 \times 6 \times 12$. For coarse partitions, we have the choice between virtually mirroring the periodic box to make the near contribution smoother, or putting more work into the near contribution. For $\epsilon = 10^{-5}$ and the given box size $1 \times 1 \times 1$, the former option turns out to be preferable. For these smaller problems we use partitions $\pi_3 = 6 \times 6 \times 6$, $\pi_4 = 6 \times 6 \times 12$, $\pi_5 = 6 \times 12 \times 12$ and $\pi_6 = 12 \times 12 \times 12$. The execution times in the graphic will again be given as $t \times \sigma/n$ for the same scaling factor $\sigma$. The execution times $t$ in the table below are given in seconds.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\pi_3$</th>
<th>$\pi_4$</th>
<th>$\pi_5$</th>
<th>$\pi_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>0.13</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>0.17</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>$2^{12}$</td>
<td>0.27</td>
<td>0.35</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>$2^{13}$</td>
<td>0.60</td>
<td>0.62</td>
<td>0.80</td>
<td>*</td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>1.57</td>
<td>1.20</td>
<td>1.17</td>
<td>1.65</td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>*</td>
<td>3.17</td>
<td>2.45</td>
<td>2.47</td>
</tr>
</tbody>
</table>

We summarize the measurement of the execution times for $2^{10} \leq n \leq 2^{20}$ in the graphic below, which displays the scaled execution times $t_\sigma = t \times \sigma/n$ to make the time complexity visible.
We can see that the experimental time complexity of MMM is more or less $O(n)$ in the range $2^{11} \leq n \leq 2^{20}$, and that there are slight irregularities in the transitions between partitions $6 \times 6 \times 6 \rightarrow 6 \times 6 \times 12$ and between partitions $12 \times 12 \times 12 \rightarrow 12 \times 12 \times 24$. This happens because the first reduction of a $1 \times 1 \times 1$ box requires more exponential terms than the next two reductions together, so this explains why the partitions $\pi_4$ and $\pi_7$ are less successful than their neighbors. The general trend of increasing $t \times \sigma/n$ may be partly explained by the $O(n \log n)$ complexity. Another important factor is the amount of memory required, which grows linearly with $n$. In any case, MMM can be seen to be efficient at least for the range $2^{10} \leq n \leq 2^{20}$.

For even smaller problems $n < 2^{10}$ it would be possible to compute several levels of virtual mirror boxes, but the current implementation of MMM does not incorporate this feature. However, we expect that $t \times \sigma/n$ will be $< 1$ for such an implementation even for very small $n$.

### 8.4 Comparison to Ewald

An alternative to MMM for small problems is the Ewald sum mentioned in section 5. If the cutoff radius is allowed to decrease with increasing $n$, Ewald’s method has an overall time complexity of $O(n^{3/2})$. Although the Ewald sum is worse in terms of complexity, it is competitive for problems $n \leq 2^{10}$ as the table below shows. To obtain a fair comparison to MMM, we have efficiently implemented Ewald’s method by applying simple optimization techniques. The execution times $t$ in the table are given in milliseconds, and the efficiency measure $t_\omega = t \times \omega / n^{3/2}$ indicates the deviation from the expected time complexity $O(n^{3/2})$. The graphic uses the time scale $t_\sigma = t \times \sigma/n$ which was introduced earlier for MMM, but note that the $t_\sigma$ axis now starts at $t_\sigma = 0$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$t$ [ms]</th>
<th>$t_\omega$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^5$</td>
<td>0.6</td>
<td>1.45</td>
</tr>
<tr>
<td>$2^6$</td>
<td>1.5</td>
<td>1.23</td>
</tr>
<tr>
<td>$2^7$</td>
<td>3.8</td>
<td>1.13</td>
</tr>
<tr>
<td>$2^8$</td>
<td>10.2</td>
<td>1.08</td>
</tr>
<tr>
<td>$2^9$</td>
<td>26.8</td>
<td>1.00</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>89.7</td>
<td>1.18</td>
</tr>
</tbody>
</table>

- $\pi_{10} = 24 \times 24 \times 48$
- $\pi_9 = 24 \times 24 \times 24$
- $\pi_8 = 12 \times 24 \times 24$
- $\pi_7 = 12 \times 12 \times 24$
- $\pi_6 = 12 \times 12 \times 12$
- $\pi_5 = 6 \times 12 \times 12$
- $\pi_4 = 6 \times 6 \times 12$
- $\pi_3 = 6 \times 6 \times 6$

Ewald sum
We see that Ewald's method is appropriate for problems of size $2^5 \leq n \leq 2^{10}$. For larger problems, the inherent time complexity $O(n^{3/2})$ of the Ewald sum shows up in addition to the increased cost due to increased memory requirements. On the other side of the spectrum, the current implementation clearly deviates from the theoretical complexity $O(n^{3/2})$, as can be seen from $t_\omega \approx 1.45$ for $n = 2^5$. This is because the benefit of optimizing loops vanishes for very small problems. In any case, together with MMM for problems of size $n > 2^{10}$, we have efficient methods to compute the Coulomb potential $\phi$ with periodic boundary conditions for a relatively wide range of problem sizes.
Conclusions

In this final section we summarize some of the results of earlier sections. Furthermore, we mention a few disadvantages of MMM compared to other approaches, and we speculate on which parts of MMM can be considered useful and which ones will remain part of a nice research project.

Of course, the most important aspect of MMM is its efficiency, but other properties are less favorable. We have compiled below the most important facets of MMM, its competitors and some practical questions.

Efficiency. The time complexity to compute the Coulomb interaction for homogeneous problems with \( m \) particles is \( O(m) \) for FMM, \( O(m \log m) \) for both PPPM and MMM, and \( O(m^{3/2}) \) for the Ewald summation. This implies that Ewald is competitive only for small problems, and we have found that the execution time for Ewald is better than MMM for about \( m \leq 10^3 \). Although the time complexities of PPPM and MMM are identical, their preferred domains are quite different. PPPM is more efficient if the required accuracy is relatively low, whereas MMM is able to obtain more accurate results within a reasonable amount of time.

We can make the above conclusions with regard to Ewald’s method and PPPM for two reasons. First, we have implemented efficient programs for Ewald and PPPM, so we can justify a comparison based on execution times. Second, the qualitative part of the conclusion remains valid more or less independent of the quality of the actual implementation. Ewald’s method is better than MMM for small problems and PPPM is better for large problems and low accuracy.

The situation is different with respect to FMM. A first obstacle is that FMM is primarily suited for computing the non-periodic Coulomb potential, although it can be applied for the periodic Coulomb potential as well. Second, existing implementations are not necessarily implemented as efficiently as MMM, so a comparison based on execution times would be unfair, unless we are only interested in the speed of actual programs. For very large \( m \), it is clear from the time complexity \( O(m) \) for FMM compared to \( O(m \log m) \) for MMM that FMM will be faster. On the other hand, our unproved impression from [4] and [5] is that MMM is faster than FMM at least for homogeneous problems with \( m \leq 10^5 \).

Accuracy. In contrast to PPPM, we can compute the periodic Coulomb interaction with MMM to arbitrary precision within reasonable time. Note however that our overall error \( \varepsilon \) defined by

\[
\varepsilon^2 = \frac{\sum_i \| \mathbf{F}_i - \mathbf{F}_i \|^2}{\sum_i \| \mathbf{F}_i \|^2}
\]

is not as innocent as it seems. It neglects other important aspects which have some influence on the attained or desired accuracy.

First, computing the Coulomb interactions with periodic boundary conditions is usually only an approximation to a real problem. Periodicity is often used to simulate a huge number of particles with a small, hopefully representative, subset of particles. Then, the
particles themselves may represent a collection of particles which are lumped together in some way. Furthermore, the interaction may only be approximately Coulombic. All these approximations introduce a discretization error, and it is clearly useless to compute the idealized periodic Coulomb interaction more accurately than imposed by the discretization error.

Second, MMM introduces a systematic error into the computation of the Coulomb interaction by partitioning the space into small boxes. For particles close to each other, the interaction is computed by using a pairwise formula and a smooth correction for mirror images. On the other hand, the interaction for distant particles is computed via an exponential expansion. Consequently, there is a small discontinuity of the potential and forces on the boundaries of the small boxes. A possible implication of these discontinuities is that we might have to compute the interaction more accurately only to make the discontinuities smaller. Furthermore, both the artificial periodic boundary conditions and the partitioning into small boxes introduce some anisotropy into the problem. The far range potential in PPPM has the same weakness, but the near range part can be made isotropic by using a cutoff radius instead of using a criterion on the box level.

Programming. We have shown that it is feasible to manually optimize important parts of MMM, as long as we restrict ourselves to a single processor. Our impression is that the optimization of routines with a narrow interface like computing $1/\sqrt{x}$ can be done within a reasonable amount of time for other processors as well. With some more work, it is even imaginable to implement all manually optimized routines efficiently on other processors.

Unfortunately, the work does not stop at that point. The code for MMM is relatively complex, and we do not expect that it can be applied usefully out of the box, that is without changes in some interfaces and data structures. The overall size of MMM is about 20000 lines of C code, if we include the code for the component tests as well. For each optimized routine, we usually have one or two functionally equivalent, but non-optimized routines. Although we consider it an integral part of the implementation, the size of MMM without the testing code would be about 10000 lines of C code. Hence, the main source of the complexity of MMM is not its size, but the scattered computation of the exponential expansion. Consider that the computation of an apparently simple term $e^{x-z}$ happens in three phases with a couple of additional combining and scaling steps in between. Although it is not easy to quantify, the ability to adapt some piece of code is actually a good indicator of its complexity. We should not delude ourselves and think that it is sufficient for MMM to run efficiently, and ignore the very relevant complexity aspect.

We have described in detail the manual optimization of elementary functions and simple subproblems. On the other hand, the overall structure of the MMM algorithm was illustrated mostly in symbolic or graphical form. A possible disadvantage of treating these problem areas in different ways is that the manual optimization part looks harder than it really was, and the overall structure part looks deceptively simple. The small subproblems can be solved and tested independently, and we have consequently done so, if possible. Therefore, we are confident that this part is not the ultimate problem for
efficient implementations of MMM. The implementation of the overall structure, on the other hand, cannot be divided easily into small pieces. This complexity is all the more inconvenient if we consider that future changes in hardware, the most important one being the ability to *preload* data in a flexible way, require changes in this complex part of MMM.

It is probably obvious by now that we do not expect MMM to immediately solve lots of problems. However, we certainly hope that MMM *creates* lots of them.
A Floating point

In this section we describe briefly the IEEE 754 floating-point standard, which is reprinted in [14] for instance. The purpose if this description is twofold. First, we need some common terminology and a precise description of the double precision floating point format in the text. Second, it is important to recognize that the IEEE standard encompasses much more than just the formats. Particularly, it contains rounding rules, special numbers like NaN and ±∞, and more.

A.1 IEEE standard

This subsection does not contain the complete reference of IEEE 754, but simply a compilation of the most relevant points of the standard.

Objectives. The standard specifies floating-point number formats, floating-point operations like add, multiply, divide, square root, conversions between integer and floating-point and between different floating-point formats, floating-point exceptions including non-numbers. It is intended that an implementation may be realized in any combination of software and hardware.

Definitions. A denormalized number is a floating-point number whose exponent is minimal for the particular format and whose leading significand bit is zero. The exponent $e$ is the component of a floating-point number that determines the factor $2^e$ for the represented number. The fraction is the field of the significand that lies to the right of the implied binary point. The significand is the component of a floating-point number consisting of a leading bit to the left of its binary point and a fraction field to the right.

Formats. The standard defines four floating-point formats single, single extended, double and double extended. The following table shows the format parameters for the four formats, blank entries representing unspecified parameters

<table>
<thead>
<tr>
<th></th>
<th>single</th>
<th>single ext.</th>
<th>double</th>
<th>double ext.</th>
</tr>
</thead>
<tbody>
<tr>
<td>#bit significand</td>
<td>$p$</td>
<td>$\geq 32$</td>
<td>$53$</td>
<td>$\geq 64$</td>
</tr>
<tr>
<td>max exponent</td>
<td>$e_{\text{max}}$</td>
<td>$\geq +1023$</td>
<td>$+1023$</td>
<td>$\geq +16383$</td>
</tr>
<tr>
<td>min exponent</td>
<td>$e_{\text{min}}$</td>
<td>$\leq -1022$</td>
<td>$-1022$</td>
<td>$\leq -16382$</td>
</tr>
<tr>
<td>bias exponent</td>
<td>$+127$</td>
<td>$+1023$</td>
<td>$+1023$</td>
<td>$+1023$</td>
</tr>
<tr>
<td>#bit exponent</td>
<td>$8$</td>
<td>$\geq 11$</td>
<td>$11$</td>
<td>$\geq 15$</td>
</tr>
<tr>
<td>#bit format</td>
<td>$32$</td>
<td>$\geq 43$</td>
<td>$64$</td>
<td>$\geq 79$</td>
</tr>
</tbody>
</table>

Within each format the following entities are provided

(i) Numbers of the form $(-1)^s \times 2^e \times b_0 b_1 b_2 \cdots b_{p-1}$ with $e_{\text{min}} \leq e \leq e_{\text{max}}$.

(ii) Two infinities $+\infty$ and $-\infty$.

(iii) At least one signaling NaN ("not a number") and one quiet NaN.

For illustration we describe the double format in detail. A 64-bit double format number $x_{(63,0)}$ contains the sign bit $s = x_{(63)}$, the biased exponent $e = x_{(62,52)}$ and the fraction $f = x_{(51,0)}$. The value $x$ of $x_{(63,0)}$ is then given by
(i) If $0 < e < 2047$, then $x = (-1)^{s}2^{e-1023} \times 1.f$

(ii) If $e = 0$ and $f = 0$, then $x = (-1)^{s} \times 0$

(iii) If $e = 0$ and $f \neq 0$, then $x = (-1)^{s}2^{-1022} \times 0.f$

(iv) If $e = 2047$ and $f = 0$, then $x = (-1)^{s} \times \infty$

(v) If $e = 2047$ and $f \neq 0$, then $x = \text{NaN}$ regardless of $s$

**Rounding.** Rounding takes a number regarded as infinitely precise and, if necessary, modifies it to fit in the desired format. Every operation specified below should be performed as if it first produced an intermediate infinitely precise result, and then rounded that result according to one of the user-selectable rounding modes round to nearest, round toward $+\infty$, round toward $-\infty$ or round toward $0$. In the round to nearest mode the representable value nearest to the infinitely precise result is to be delivered. In case of two nearest representable values, the one with its least significant bit zero is to be delivered.

**Operations.** All implementations of the standard shall provide operations to add, subtract, multiply, divide, compare, extract the square root, find the remainder, round to integer, convert between different floating-point formats, convert between floating-point and integer formats and convert from binary to decimal and vice versa. Except for the binary $\leftrightarrow$ decimal conversion, each of the operations shall be performed with exact rounding. Special rules are given for arithmetic involving $\pm \infty$, NaN, and sometimes 0.

**Exceptions.** There are five types of exceptions which have to be signaled, which entails setting a status flag, taking a trap or possibly doing both. Specified exceptions are invalid operation like taking $\sqrt{-1}$, division by zero, overflow, underflow and inexact. A user should be able to request a trap on any of the five exceptions by specifying a trap handler.

**A.2 Comments**

Computing division and square root with correct rounding is significantly more expensive than computing an approximation to the desired result, which is correct except for one or two least significant bits. The situation for a software-based solution is even more dramatic, where it is much more expensive to compute exactly rounded results for operations like division or square root. Consequently, routines to compute square roots in software usually sacrifice IEEE compliance to remain reasonably efficient. In some cases described in the text we have gone even farther by ignoring NaN and $\pm \infty$, for instance, based on the assumption that these cases will never occur. Furthermore, we have simply neglected the existence of denormalized numbers where it would have been awkward to handle them. To some extent, the additional cost for rounding and the necessity of such assumptions illustrate that the requirements of high-performance implementations in both hardware and software might be given more weight in a future floating-point standard.
B Alpha 21164

This section summarizes the most important characteristics of the 21164 Alpha microprocessor, at least those which are relevant for this text. The 21164 implements the Alpha architecture, a RISC architecture designed for high performance. Before we describe the microprocessor, a quick overview of the architecture is given.

B.1 Alpha architecture

Note that this is only a sketch of the architecture. Consult the Alpha architecture reference manual [6] for details.

Registers. The Alpha architecture is a 64-bit load and store RISC architecture. Both integer and floating point registers are 64 bit long and all operations are performed between 64-bit registers.

Instructions. Instructions are always 32 bits long, thereby making multiple instruction issue simpler. Memory operations are either load or store operations, and all data manipulation is done between registers. Instructions interact with each other only via register or memory locations, i.e., one instruction writes a datum and another instructions reads this datum. There are no condition codes set implicitly by an operation, for instance.

Addresses. The basic addressable unit in the Alpha architecture is the 8-bit byte. The 21164 supports a 43-bit virtual address and a 40-bit virtual address space, for instance.

Data types. The Alpha architecture supports four integer data types, namely the byte (8 bit), word (16 bit), longword (32 bit) and quadword (64 bit). The starting address of an integral datum may be arbitrary, but Alpha implementations may impose a significant performance penalty when accessing operands which are not naturally aligned. The Alpha architecture supports the IEEE single precision (32 bit) and double precision (64 bit) formats as well as VAX architecture 32-bit and 64-bit floating point formats.

B.2 21164 Microprocessor

The reason we are interested in so much detail about the 21164 is that we want to write code which executes as quickly as possible. This goal can only be achieved by looking at aspects like issue rules, instruction scheduling, cache characteristics and so forth. In the following we give a short summary of the most important things to consider, see [7] for details.

Instruction issue. The instruction decode unit decodes up to four instructions $i_1 \ldots i_4$ in parallel and checks if the required resources—an operand in a register or a functional unit for instance—are available for each instruction. The decode unit then issues the initial sequence of instructions $i_1 \ldots i_n$ for which all required resources are available. Instructions are not issued out of order, so $i_3$ will not be issued before $i_2$ even if the resources for $i_3$ are available. Furthermore, the decode unit does not proceed until all of $i_1 \ldots i_4$ are issued, only then a new group of four instructions is decoded. Thus, achieving optimal performance requires that code be scheduled properly and that nop instructions be used to fill empty slots.
**Integer execution unit.** The integer execution unit contains two 64-bit integer execution pipelines, E0 and E1. Among other things, the pipelines include two adders, two logic boxes and one barrel shifter. The unit also contains 32 integer registers in a register file with four read ports and two write ports. The write ports accept results from both pipes and load instruction results. The 21164 is able to issue two integer instructions per cycle, one issue for E0 and E1 each.

**Floating point execution unit.** The on-chip floating point unit can execute both IEEE and VAX floating point instructions. The most important features of the floating point unit are the 32 floating point registers, each 64 bits wide, a floating point multiply pipeline FM and a floating point add pipeline FA. The unit can accept one multiply and one add each cycle, whereas the floating point divide unit is associated with the add pipeline, but is not pipelined. The floating point register file has five read ports and four write ports. This number of ports is required for a peak performance of two floating point operations per cycle in addition to two loads or one store.

**Memory unit.** We will not describe the address translation here, but mention another important feature. If the addressed location of a load is found in the first-level cache, then the required value is written to either a floating point register or integer register. If the data is not found in the cache, then the miss address file is consulted to see if it contains a load which addresses the same cache line. If it does, then the new load—certain conditions being satisfied—is merged with an existing miss address file entry. This allows for servicing two or more misses with one data fill. There are six miss address file entries and four more for instruction unit fetches, so loads into up to ten cache lines may be outstanding. Similarly, the write buffer is able to combine writes to the same cache line, and contains six 32-byte entries. The ability to merge loads or stores into one entry means that sequential memory accesses will not fill load and store buffers unnecessarily.

**Cache organization.** The 21164 has three on-chip caches, a primary data cache, a primary instruction cache, and a second-level cache for both data and instructions. The data cache is a dual-read-ported, single-write-ported, 8KB cache. It is a write-through, read-allocate, direct-mapped, byte-accessible, physical cache with 32-byte blocks. The instruction cache is a virtual, direct-mapped 8KB cache with 32-byte blocks. The second-level cache is a 96KB, 3-way, set-associative, physical, write-back, write-allocate, byte-accessible cache with 32-byte or 64-byte blocks. The second-level cache is fully pipelined and processes read and write operations at the rate of one 16-byte chunk per cycle.

**Issue rules.** The four pipelines E0, E1, FA and FM of the 21164 allow issues of different instruction classes. The following table shows this relationship in a simplified—and hence slightly incorrect—form. Note that some instructions are not fully pipelined, integer multiplication and floating-point divide, for instance. Instructions which are usually excluded from the issue analysis in the text are shown like this.
In addition to the restrictions due to available functional units depicted above, other conditions may prevent the 21164 from issuing an instruction. The most important rules are listed below:

(i) No instruction can be issued until all of its source and destination registers are clean, i.e., there are no outstanding writes to the source or destination registers.

(ii) No load can be issued simultaneously with a store or in the second cycle after a store has been issued.

(iii) Several restrictions implied by non-pipelined operations apply. No integer multiplication can be issued if the multiplier is busy and no instruction can be issued to pipe E0 exactly two cycles before an integer multiply completes. Similar restrictions exist for the floating-point divide.

(iv) No instruction can be issued to pipes E0 or E1 two cycles before an integer register fill is requested by the cache unit. And no load or store may be issued one cycle before such an integer register fill request.

**Instruction latencies.** We may differentiate between two types of latencies. The issue latency tells us how many cycles we have to wait until we may issue another instruction of the same type. The result latency gives the number of cycles required until the result is available. For a fully pipelined functional unit, the issue latency is 1, but the result latency is usually >1 except for simple operations. The following table contains the latency information for the most important instruction classes. Note that those are idealized numbers because restrictions like finite load or store buffers are not taken into account. An issue latency of 1/2 means that two such instructions may be issued per cycle. The result latency for branches is to be understood as the penalty in comparison to sequential code.

Note that this table is a slight simplification of the true situation. A special bypass provides an effective latency of 0 cycles for integer comparisons when executed in the same cycle as a conditional branch, for instance.
class | issue latency | result latency
load | \( = \frac{1}{2} \) | \( = 2 \) if cache hit
| | \( \geq 8 \) if cache miss
store | 1 | no result
branch | 1 | \( \geq 1 \) if branch taken
| | \( \geq 5 \) if branch mispredict
integer | \( = \frac{1}{2} \) if arithmetic/logic/move | \( = 1 \) if arithmetic/logic/shift
| | \( = 1 \) if shift | \( = 2 \) if conditional move
| | \( \geq 4 \) if multiply | \( \geq 8 \) if multiply
float | 1 if add/multiply | 4 if add/multiply
| \( \approx 20 \) if divide single prec | \( \approx 20 \) if divide single prec
| \( \approx 30 \) if divide double prec | \( \approx 30 \) if divide double prec

B.3 Implications

We summarize a few implications which may be drawn from the data above, particularly the ones relevant for our problem. First of all, it makes sense to partition the instructions into classes

- \( \bullet \) = store (E0)
- \( \odot \) = load (E0 or E1)
- \( \varnothing \) = shift (E0)
- \( \odot \) = integer op (E0 and E1) or branch (E1)
- \( \odot \) = floating-point add (FA)
- \( \odot \) = floating-point mul (FM)

which allows to schedule instructions with respect to available pipelines and with respect to the most important issue rules. Although branches can only be issued into E1, there is no harm in using the symbol \( \odot \). This is only misleading insofar as we might be tempted to schedule two branches within the same cycle, so we will have to watch out for this special case.

Since we are interested in optimizing floating-point computations, the most important characteristic of the 21164 is that the floating-point add pipeline and the floating-point multiply pipeline are fully pipelined. So we may execute one add and one multiply each cycle, and the result will be available 4 cycles later for both add and multiply. This latency implies that a loop unrolling factor of 4 will often be the natural choice. The efficiency of the add and multiply instructions is in clear contrast to the floating-point division. Not only is the division relatively slow, but it is not pipelined at all. It is easy to see that this is one of the main reasons why it is possible to compute \( \frac{1}{\sqrt{\chi}} \) in a vectorized fashion much more efficiently than scalar codes.

Unlike most other current processors, the 21164 is not dynamically scheduled, and even the issue structure may be called static. This is one of the main reasons for the high clock rate of the 21164 compared to the competitor processors. On the other hand, static scheduling means that good instruction scheduling at compile time is crucial, which cannot always be done by the compiler alone.
C  Machines

In this section, we compile some technical information for the machines we perform the experiments on. In the text, we usually designate these machines by their type of processor, namely Alpha, MIPS, UltraSPARC-II, UltraSPARC and SuperSPARC. The technical information includes the model, the CPU and the clock rate, the cache and memory sizes, the operating system, and compiler versions and flags used. Depending on the problem, the compiler may or may not simplify floating point expressions, so we have adjusted the compile options, if necessary.

C.1  Alpha 21164

<table>
<thead>
<tr>
<th>model</th>
<th>Digital AlphaStation 500/500</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Alpha 21164A EV5.6</td>
</tr>
<tr>
<td>clock rate</td>
<td>500 MHz</td>
</tr>
<tr>
<td>on-chip cache</td>
<td>primary: 8K data + 8K instruction</td>
</tr>
<tr>
<td></td>
<td>secondary: 96K 3-way data and instruction</td>
</tr>
<tr>
<td>external cache</td>
<td>8M data and instruction</td>
</tr>
<tr>
<td>memory</td>
<td>512M</td>
</tr>
<tr>
<td>operating system</td>
<td>Digital Unix V4.0d</td>
</tr>
<tr>
<td>cc compiler</td>
<td>Digital UNIX Compiler Driver 3.11</td>
</tr>
<tr>
<td>cc flags</td>
<td>cc -tune host -fast or</td>
</tr>
<tr>
<td></td>
<td>cc -tune host -fast -no_fp_reorder</td>
</tr>
<tr>
<td>gcc compiler</td>
<td>gcc version 2.7.2.1</td>
</tr>
<tr>
<td>gcc flags</td>
<td>gcc -Wall -03 or</td>
</tr>
<tr>
<td></td>
<td>gcc -Wall -01</td>
</tr>
<tr>
<td>f77 compiler</td>
<td>Digital Fortran V5.0-138</td>
</tr>
<tr>
<td>f77 flags</td>
<td>f77 -tune host -fast</td>
</tr>
</tbody>
</table>

C.2  MIPS R10000

<table>
<thead>
<tr>
<th>model</th>
<th>SGI Octane/SI (IP30)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>MIPS R10000</td>
</tr>
<tr>
<td>clock rate</td>
<td>195 MHz</td>
</tr>
<tr>
<td>on-chip cache</td>
<td>32K 2-way data + 32K 2-way instruction</td>
</tr>
<tr>
<td>external cache</td>
<td>1M data and instruction</td>
</tr>
<tr>
<td>memory</td>
<td>256M</td>
</tr>
<tr>
<td>operating system</td>
<td>Irix 6.4</td>
</tr>
<tr>
<td>cc compiler</td>
<td>MIPSpro Compilers: Version 7.20</td>
</tr>
<tr>
<td>cc flags</td>
<td>cc -64 -0fast=ip30 or</td>
</tr>
<tr>
<td></td>
<td>cc ... -OPT:fold.reassociate=OFF</td>
</tr>
<tr>
<td>gcc compiler</td>
<td>gcc version 2.95 19990728 (release)</td>
</tr>
<tr>
<td>gcc flags</td>
<td>gcc -Wall -03 or</td>
</tr>
<tr>
<td></td>
<td>gcc -Wall -01</td>
</tr>
<tr>
<td>f77 compiler</td>
<td>MIPSpro Compilers: Version 7.20</td>
</tr>
<tr>
<td>f77 flags</td>
<td>f77 -64 -0fast=ip30</td>
</tr>
</tbody>
</table>
### C.3 UltraSPARC-II

The UltraSPARC-II and SuperSPARC are multiprocessor machines, but the execution times in the text are still given for a single processor. For these multiprocessor machines, the cache size is given per processor, but the memory size is for the machine, that is for all processors together.

<table>
<thead>
<tr>
<th></th>
<th>Sun Enterprise 3500</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>6× UltraSPARC-II</td>
</tr>
<tr>
<td>clock rate</td>
<td>336 MHz</td>
</tr>
<tr>
<td>on-chip cache</td>
<td>16K data + 16K instruction</td>
</tr>
<tr>
<td>external cache</td>
<td>4M data and instruction</td>
</tr>
<tr>
<td>memory</td>
<td>3072M</td>
</tr>
<tr>
<td>operating system</td>
<td>Solaris 2.6</td>
</tr>
<tr>
<td>cc compiler</td>
<td>WorkShop Compilers 5.0 98/12/15 C 5.0</td>
</tr>
<tr>
<td>cc flags</td>
<td>cc -native -fast</td>
</tr>
<tr>
<td>gcc compiler</td>
<td>gcc version 2.95.1 19990816 (release)</td>
</tr>
<tr>
<td>gcc flags</td>
<td>gcc -Wall -03 or gcc -Wall -01</td>
</tr>
<tr>
<td>f77 compiler</td>
<td>WorkShop Compilers 5.0 99/03/24</td>
</tr>
<tr>
<td>f77 flags</td>
<td>f77 -native -fast</td>
</tr>
</tbody>
</table>

### C.4 UltraSPARC

<table>
<thead>
<tr>
<th></th>
<th>Sun Ultra 1/140</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>UltraSPARC</td>
</tr>
<tr>
<td>clock rate</td>
<td>143 MHz</td>
</tr>
<tr>
<td>on-chip cache</td>
<td>16K data + 16K instruction</td>
</tr>
<tr>
<td>external cache</td>
<td>512K data and instruction</td>
</tr>
<tr>
<td>memory</td>
<td>128M</td>
</tr>
</tbody>
</table>

The operating system, compiler versions and compile flags for the UltraSPARC are identical to the UltraSPARC-II.

### C.5 SuperSPARC

<table>
<thead>
<tr>
<th></th>
<th>Sun SPARCstation 20/514</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>4× SuperSPARC</td>
</tr>
<tr>
<td>clock rate</td>
<td>50 MHz</td>
</tr>
<tr>
<td>on-chip cache</td>
<td>16K data + 20K instruction</td>
</tr>
<tr>
<td>external cache</td>
<td>1M data and instruction</td>
</tr>
<tr>
<td>memory</td>
<td>512M</td>
</tr>
</tbody>
</table>

The operating system, compiler versions and compile flags for the SuperSPARC are identical to the UltraSPARC-II.
D Tools

This section describes two tools we have repeatedly used for generating code and for writing loop unrolled code painlessly. The emphasis in these tools is both in simplicity and functionality. An advantage of writing this type of programs yourself is that you have something at hand which fits your needs, and which can still be easily modified due to its simplicity.

D.1 cdup — code duplication

The basic idea of this tool is best explained with an example, which is only a little simpler than a real problem. Assume you want to implement the daxpy routine yourself, then a straightforward implementation of the loop in C might look like

```
for (n = 0; n < nof; n++) {
    y[n] += alpha*x[n];
}
```

Usually the compiler will be able to optimize this code by loop unrolling, given that it recognizes that the arrays x[] and y[] do not overlap. If the compiler cannot determine this at compile time, it can still check for this property at runtime and generate two variants of the code, an efficient loop unrolled version for disjoint x[] and y[], and a slow sequential version for overlapping x[] and y[]. Another possibility is to unroll the loop yourself and making sure that instruction scheduling is made easy for the compiler, for instance by the following code

```
for (n = 0; n < nof; n += 4) {
    y0 = y[n+0]; x0 = x[n+0];
    y1 = y[n+1]; x1 = x[n+1];
    y2 = y[n+2]; x2 = x[n+2];
    y3 = y[n+3]; x3 = x[n+3];
    y[n+0] = y0 + alpha*x0;
    y[n+1] = y1 + alpha*x1;
    y[n+2] = y2 + alpha*x2;
    y[n+3] = y3 + alpha*x3;
}
```

where we have assumed that nof is divisible by 4. Note that this transformation is trivial and consists mainly in typing instructions which look nearly the same. While this is not intellectually difficult, this weak form of code duplication is still painful, especially if one wants to try several loop unrolling factors and variants involving software pipelining. A more convenient way to write the above was found to be

```
for (n = 0; n < nof; n += 4) {
    #4 y@ = y[n+@]; x@ = x[n+@];
    #4 y[n+@] = y@ + alpha*x@;
}
```
with identical semantics. In this example, the @ character is simply replaced by 0, 1, 2 and 3 to generate several lines of code.

The impressions one may gain from this little example are that the transformation from the @ representation is trivial, and that it might be similarly easy to write the expanded code directly. The former impression is correct, but the latter one underestimates the tedious work involved. Let us demonstrate this with an excerpt from a piece of code which computes

\[ y_n \rightarrow y'_n = y_{n-1} + y_n + y_{n+1} \]

with periodic boundary conditions. The excerpt of the associated @ code looks

```c
#= (0,1,4,5)
... for (n = 2; n < nof-2; n += 2) {
...  ## c0 = xc[0]; d0 = xc[0+2];
  ## b0 = b0 + c0;
  ## t0 = a0 + b0;
  ## xa[0+2] = t0;
  t0 = b0 + d0; a0 = c0;
  ## xc[0] = t0;
... }
... }
```

Note that this computation is complicated by the fact that the data y is not stored sequentially in memory but in small 2 \times 2 \times 2 blocks to exploit cache lines. The corresponding expanded code then looks

```c
... for (n = 2; n < nof-2; n += 2) {
...  c0 = xc[0]; d0 = xc[0+2];
  c1 = xc[1]; d1 = xc[1+2];
  c4 = xc[4]; d4 = xc[4+2];
  c5 = xc[5]; d5 = xc[5+2];
  b0 = b0 + c0;
  b1 = b1 + c1;
  b4 = b4 + c4;
  b5 = b5 + c5;
  t0 = a0 + b0;
  t1 = a1 + b1;
  t4 = a4 + b4;
  t5 = a5 + b5;
  xa[0+2] = t0; t0 = b0 + d0; a0 = c0;
  xa[1+2] = t1; t1 = b1 + d1; a1 = c1;
  xa[4+2] = t4; t4 = b4 + d4; a4 = c4;
... }
... }
```
It is not a coincidence that we have taken one of the combining routines \( \sigma \rightarrow \Sigma \) to illustrate this tool. For efficiency reasons it is a good idea to implement different ways to combine values in separate routines, so this tool has made this task a lot easier.

The main drawback of this approach is that it allows you to express only very regularly structured types of loop unrolling and software pipelining. A more fine-grained approach is shown in the following subsection.

### D.2 cgen — code generation

The cgen tool is appropriate if you want to prescribe the exact spot in the instruction stream for each instruction. Additionally, a simple scheme supports code generation for particular types of optimization, especially loop unrolling and software pipelining. The code duplication implied by these optimization techniques is made relatively painless by this tool, in some sense similar to the cdup tool described above.

To illustrate this tool we use the simple daxpy example. Since iterations are mutually independent, loop unrolling is easy and corresponds to executing a few instances of the loop in parallel. The software pipelining structure is given by the following algorithm:

```plaintext
function daxpy =
  x = x_1, y = y_1
  s = \alpha x
  for k = 2 \ldots n do
    t = y + s, x = x_k, y = y_k
    y_{k-1} = t, s = \alpha x
  od
  t = y + s
  y_n = t
```

where we have not yet considered the loop code. The loop code consists of instructions to update pointers to the arrays \( x_k \rightarrow x_{k+1}, y_k \rightarrow y_{k+1} \), counting the number of iterations \( k = k + 1 \) and of course the conditional branch, which contribute 4 \( \Diamond \) per loop. In the issue map the loop unrolling and software pipelining steps then look as follows:

![Issue Map](image)
To make the description more explicit, the following table shows in detail how the instruction sequence is assembled to obtain the above issue map. Since we have fourfold unrolled and twofold pipelined the loop, each instruction will occur in \(4 \times 2 = 8\) spots in the instruction sequence. In the graphical representation below, we have shown the 4 instances of the loop unrolled code explicitly.

<table>
<thead>
<tr>
<th>instruction</th>
<th>issue spots</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = x_k)</td>
<td><img src="image1" alt="Issue spots for (x = x_k)" /></td>
</tr>
<tr>
<td>(y = y_k)</td>
<td><img src="image2" alt="Issue spots for (y = y_k)" /></td>
</tr>
<tr>
<td>(s = \alpha x)</td>
<td><img src="image3" alt="Issue spots for (s = \alpha x)" /></td>
</tr>
<tr>
<td>(t = y + s)</td>
<td><img src="image4" alt="Issue spots for (t = y + s)" /></td>
</tr>
<tr>
<td>(y_k = t)</td>
<td><img src="image5" alt="Issue spots for (y_k = t)" /></td>
</tr>
<tr>
<td>(x_k \rightarrow x_{k+1})</td>
<td><img src="image6" alt="Issue spots for (x_k \rightarrow x_{k+1})" /></td>
</tr>
<tr>
<td>(y_k \rightarrow y_{k+1})</td>
<td><img src="image7" alt="Issue spots for (y_k \rightarrow y_{k+1})" /></td>
</tr>
<tr>
<td>(k = k + 1)</td>
<td><img src="image8" alt="Issue spots for (k = k + 1)" /></td>
</tr>
<tr>
<td>if (k &lt; n) loop</td>
<td><img src="image9" alt="Issue spots for if (k &lt; n) loop" /></td>
</tr>
</tbody>
</table>

Facilitating this type of instruction placement and duplication is exactly what the tool cgen is good for. In perl readable form, the above table suited for generating C code might look like

```perl
$seqn = [
    [19], NONE, ['loop:'],
    [24], INT, ['x += 4;'],
    [25], INT, ['y += 4;'],
    [26], INT, ['nof -= 4;'],
    [27], BR, ['if (nof) goto loop;'],
    [10..13, 20..23], LD, ['x$m = x[$ldx];'],
    [10..13, 20..23], LD, ['y$m = y[$ldy];'],
    [12..15, 24..27], MUL, ['s$m = alpha*x$m;'],
    [20..23, 30..33], ADD, ['t$m = y$m + s$m;'],
    [24..27, 34..37], ST, ['y[$sty] = t$m;'],
];
```

Note that the instructions in the pre phase are numbered starting at 10, those in the loop at 20 and those in the post phase starting at 30. The perl variable \$m designates the instance of the iteration modulo 4, and the variables \$ldx, \$ldy and \$sty contain the appropriate pointer offsets. Together with the declarations and a few additional instructions, the generated code for this simple example then looks...
void daxpy_s (int nof, double alpha, double *x, double *y) {
    double x0, y0, s0, t0;
    double x1, y1, s1, t1;
    double x2, y2, s2, t2;
    double x3, y3, s3, t3;

    ASSERT(((nof & 0x03) == 0) && (nof >= 8));
    nof -= 4;
    X(010) x0 = x[0]; y0 = y[0]; FNOP; FNOP;
    X(011) x1 = x[1]; y1 = y[1]; FNOP; FNOP;
    X(012) s0 = alpha * x0; y2 = y[2]; x2 = x[2]; FNOP;
    X(013) s1 = alpha * x1; y3 = y[3]; x3 = x[3]; FNOP;
    X(014) s2 = alpha * x2; NOP; NOP; FNOP;
    X(015) s3 = alpha * x3; NOP; NOP; FNOP;

    loop:
    X(020) t0 = y0 + s0; y0 = y[4]; x0 = x[4]; FNOP;
    X(021) t1 = y1 + s1; y1 = y[5]; x1 = x[5]; FNOP;
    X(022) t2 = y2 + s2; y2 = y[6]; x2 = x[6]; FNOP;
    X(023) t3 = y3 + s3; y3 = y[7]; x3 = x[7]; FNOP;
    X(024) y[0] = t0; s0 = alpha * x0; x += 4; FNOP;
    X(025) y[1] = t1; s1 = alpha * x1; y += 4; FNOP;
    X(026) y[-2] = t2; s2 = alpha * x2; nof -= 4; FNOP;
    X(027) y[-1] = t3; s3 = alpha * x3; if (nof) goto loop; FNOP;
    X(030) t0 = y0 + s0; NOP; NOP; FNOP;
    X(031) t1 = y1 + s1; NOP; NOP; FNOP;
    X(032) t2 = y2 + s2; NOP; NOP; FNOP;
    X(033) t3 = y3 + s3; NOP; NOP; FNOP;
    X(034) y[0] = t0; NOP; FNOP; FNOP;
    X(035) y[1] = t1; NOP; FNOP; FNOP;
    X(036) y[2] = t2; NOP; FNOP; FNOP;
    X(037) y[3] = t3; NOP; FNOP; FNOP;
}

Note that instruction numbers like X(033) indicating the instruction placement will be removed by the preprocessor and are irrelevant for the compiler. The tokens NOP and FNOP insert integer and float nop instructions into the instruction stream, respectively. This can be easily done with gcc, which allows to insert assembler instructions into C code in a convenient way. The other vital feature of gcc is that the amount and type of optimizations applied may be restricted appropriately. In our case we are interested in letting the compiler do the register assignment, but we do not want the compiler to perform instruction scheduling or even loop optimizations. Of course, this strategy only works if one C statement corresponds to exactly one assembler instruction, otherwise the desired structure of 4 instructions being issued at the same cycle is destroyed.
E Compilers

In this section we briefly investigate the optimizing capabilities of the C and Fortran compilers on the Alpha 21164. First, we compare the execution times for two key algorithms compiled by the optimizing C and Fortran compilers as well as for a manually optimized version of the code. Second, we perform a crude analysis of the generated codes to find out why the manually optimized code remains superior.

E.1 Timings

We have seen in several places in the text that manual optimization often improves the execution time significantly, if we compare it to a sequential variant of the code compiled by the C compiler. One purpose of this subsection is to find out if the Fortran compiler can do better than this. The algorithms we use for the comparison are the routines for computing $1/\sqrt{x}$ and the procedure $\phi_{a0}$ required in the phase $\Sigma \rightarrow \phi$. Both routines contribute significantly to the cost of computing the potential $\phi$ for a problem with periodic boundary conditions.

The $1/\sqrt{x}$ routine is interesting because there are substantial amounts of both integer and floating point instructions, which may be executed simultaneously if scheduled correctly. We assume that the vector length $n = 100$, that is we want to compute $1/\sqrt{x}$ for $n$ different values $x$ at once. The execution times in the table below are given in ns per element.

<table>
<thead>
<tr>
<th>Processor</th>
<th>$t$ [ns]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164</td>
<td>22</td>
</tr>
<tr>
<td>500 MHz</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>127</td>
</tr>
<tr>
<td></td>
<td>46</td>
</tr>
</tbody>
</table>

Obviously, the Fortran compiler was able to optimize the sequential code substantially. We will see later in the analysis that the Fortran compiler succeeded in identifying different iterations of the loop as independent. Combined with loop unrolling and subsequent instruction scheduling this leads to an improvement by about a factor 4. Compared with the manually optimized variant, however, the sequential Fortran version is still much slower.

The $\phi_{a0}$ routine has other characteristics than the $1/\sqrt{x}$ routines. It contains almost no integer operations except for a few loads and stores, and there are relatively few dependences between the floating point operations, as you may see from the dependence graph shown earlier. For the timings we used $n = 16$ for the number of particles and $m = 37$ for the number of $(p, q)$ indices required, and the execution times are given in ns for a single $s_{pq} \times \sum_{pq}$, which corresponds to 41 multiplications in the optimized algorithm.
In contrast to the $1/\sqrt{x}$ routine, the C and Fortran compilers generate equally efficient code. Since this routine consists primarily of floating point multiplications with relatively few dependences, it is no surprise that the compilers are able to optimize it fairly well. Still, by manually optimizing it we improve the execution time by about a factor of 1.5. If we compile the manually optimized code with the optimizing C compiler, we obtain execution times between the optimal variants and the sequential variants.

### E.2 Analysis

In this subsection we investigate why the manual optimizations are more successful than the compiler based optimizations. In some sense, manual optimizations will *always* be superior since we may always choose the compiler optimized version as our starting point. The interesting point is that our manual optimizations are based on simple code transformations, which could be applied by a compiler as well. Let us first look at the issue map of the optimized code to compute $1/\sqrt{x}$, which was obtained by fourfold unrolling and twofold pipelining.

Note that we have only indicated the pre and post phases, and we will ignore this overhead for the rest of this section. Furthermore, we will make no distinction between the actual code for computing $1/\sqrt{x}$ and the loop code, which would only complicate the graphics unnecessarily. The issue map for the code generated by the optimizing Fortran compiler looks

We may conclude that the Fortran compiler recognizes the independence of different iterations. Just as we did in the manual optimization, the compiler unrolls the loop fourfold, and then it schedules the instructions of the four instances of the loop more or less in parallel. Since
the compiler does not substantially overlap integer and floating point operations by applying software pipelining, the resulting issue map still consists of 79 cycles compared to 40 cycles for the manually optimized version. We see that the software pipelining step is the only important difference between the two optimization schemes in this case.

It is interesting to see what the C compiler does wrong in trying to optimize the manually optimized code. The following issue map shows the code after using the optimizing cc compiler:

![Issue Map](image1)

There is no obvious structure in the misplacement of instructions, which would indicate a particular weakness in the compiler, its scheduling is simply a little weaker. A cycle count shows that this compiler optimized version will take at least 48 cycles compared to 40 cycles for the manually optimized version. To demonstrate the importance of instruction sequence alignment, we shift the loop body by one instruction, by inserting a nop instruction before the loop, for instance. The resulting issue map then looks:

![Issue Map](image2)

which takes at least 56 cycles to execute. Consequently, the execution time will increase by about 17% compared to the unmodified code, although the instruction sequences within the loop are identical. The reason for this difference is that the Alpha 21164 can issue instructions simultaneously only if they are within the same aligned 32 bytes in the instruction stream, see appendix B for details.

Analogously, we may look at the issue maps of the routine $\phi_{d0}$, both for the manually optimized code and for the code generated by the optimizing Fortran compiler. The manual optimization consists of twofold pipelining, and the resulting issue map looks:

![Issue Map](image3)

The corresponding issue map for the code generated by the Fortran compiler is given by:

![Issue Map](image4)
The situation is fairly similar to the case $1/\sqrt{x}$ above, since the main difference between the manual and the compiler-directed optimizations is again the software pipelining step. Besides this, the Fortran compiler loses a few cycles by non-optimal scheduling. These simple differences lead to 41 cycles in the inner loop for the manually optimized code compared to 66 cycles for the Fortran generated code.

Based on these two examples we may conclude that software pipelining is a technique worth to be used more often in compilers as well. Another important point is that unstructured scheduling is not always required, since patterns like loop unrolling with independent iterations often suffice to obtain reasonable performance. In fact, unstructured scheduling was unsuccessful for the $1/\sqrt{x}$ using the cc compiler, whereas simple loop unrolling by the Fortran compiler was quite effective.
In this concluding section we will speculate about important characteristics of processors and the associated memory system. The discussion will be restricted to problems analyzed in this text, and for concreteness we look at the Alpha 21164 and Alpha 21264 microprocessors. Nevertheless, some conclusions may be relevant in a broader context. We first summarize important differences of the Alpha 21164 and Alpha 21264, and then we comment on particular features. Note that the analysis in this section is influenced by personal opinions, so your mileage may vary.

F.1 Alpha 21264

The description of the Alpha 21164 is given in appendix B, so it suffices to summarize a few key differences between the Alpha 21264 and the Alpha 21164. We will simply name the processors by their numbers 21164 and 21264 from now on.

**Instruction issue.** The instruction decode unit of the 21264 decodes up to four instructions in parallel, which end up in the integer or floating-point issue queues. From there, instructions may be issued out-of-order, and the 21264 is able to perform dynamic scheduling, register renaming and speculative execution. This is very different from the 21164, which only issues instructions in order.

**Integer execution unit.** The 21264 contains four integer units, two for integer operations and two for load and store operations. In the earlier 21164, integer operations and the address calculation for load and store operations are done in the same two integer units.

**Floating point unit.** Both the 21164 and 21264 contain fully pipelined floating point units for add and multiply.

**Memory unit.** Load and store buffers are a little larger in the 21264 than in the 21164. The main difference is in the cache structure described below.

**Cache organization.** Probably the most important difference to the 21164 is the data cache structure of the 21264. It contains a 2-way 64K on-chip cache, and it completely controls the off-chip second-level cache. This control makes it possible to establish high bandwidths between off-chip memory and the primary cache. In contrast to this, the 21164 contains first and second-level on-chip caches of 16K and 96K, respectively.

**Issue rules.** Since the 21264 reorders instructions, issue rules are not as important for scheduling code at compile time as they are for the 21164.

**Instruction latencies.** Latencies for the 21264 are similar to the latencies for the 21164. Of course, floating-point add and multiply are still fully pipelined.

These differences imply that the 21264 is less sensitive to good instruction scheduling at compile time, and—depending on the particular machine—accesses to memory outside the processor may be much faster with the 21264 than with the 21164. Of course, implementing features like dynamic scheduling has a negative impact on the clock rate. So, while we can see the old 21164 running at 600 MHz, the new 21264 steps back to 500 MHz.
F.2 Dynamic scheduling

In this subsection we discuss the usefulness of dynamic scheduling as implemented in the 21264 compared to static scheduling implemented in the 21164. The question is if it is profitable to trade clock rate versus dynamic scheduling. For our MMM application, we can see that a high clock rate is more important. Using manual optimization, we are able to statically schedule key algorithms such that the clock rate is the main limiting factor on the 21164. For this type of code, dynamic scheduling does not improve the execution time significantly, but reducing the clock rate does hurt performance. This argument applies to manually optimized, compute bound algorithms like $1/\sqrt{x}$, $e^{\pm x}$ and $\phi_{\neq 0}$. For memory bound codes like the phase $\sigma \rightarrow 2$, neither dynamic scheduling nor clock rate have a significant influence on the execution time. The essential characteristic for optimized memory bound codes is the available bandwidth. In this respect, the 21264 is promising thanks to its interface to off-chip caches or memories. Of course, this higher bandwidth is not for free, but it involves costs in terms of money and probably in terms of increased latency as well, depending on what alternatives the comparison is based on.

In our opinion, it is still an open question if statically or dynamically scheduled processors are to be preferred, although with the introduction of the 21264 the pendulum seems to swing towards the side of dynamic scheduling. Our particular application shows that this might be the wrong side. Let us look at the daxpy example to explain some issues involved in the question of dynamic versus static scheduling. A single iteration may be written

\[
daxpy_k =
\begin{align*}
x &= x_k & \quad \circ \quad \\
t &= \alpha x & \quad \circ \quad \\
y &= y_k & \quad \circ \quad \\
t &= t + y & \quad \circ \quad \\
y_k &= t & \quad \circ \quad 
\end{align*}
\]

While the particular order of these instructions is of little importance with dynamic scheduling, it is essential with static scheduling. Imagine that neither $x_k$ nor $y_k$ are in the cache, then a statically scheduled processor will stall before computing $\alpha x$, and it will only then ask for $y_k$, and consequently stall again before computing $t + y$. This is the case even if we may have several loads outstanding, as we assume from now on. For a dynamically scheduled processor, the load of $y_k$ is issued while $\alpha x$ is still waiting for $x$. Being able to issue the load of $y_k$ means that we have effectively reduced the number of stalls from two to one. Of course, a simple instruction reordering makes sure that the delays due to loading $x_k$ and $y_k$ overlap even in the statically scheduled case. The analogous code with two iterations

\[
daxpy_{k...k+1} =
\begin{align*}
x &= x_k & \quad \circ \quad \\
y &= y_k & \quad \circ \quad \\
t &= \alpha x & \quad \circ \quad \\
t &= t + y & \quad \circ \quad \\
y_k &= t & \quad \circ \quad \\
x &= x_{k+1} & \quad \circ \quad 
\end{align*}
\]
is reasonably efficient for dynamically scheduled processors. For the statically scheduled case we have to prevent the case of multiple stalls mentioned above by reordering as follows:

\[
\begin{align*}
    y &= y_{k+1} \\
    t &= \alpha x \\
    t &= t + y \\
    y_{k+1} &= t
\end{align*}
\]

Note that this scheduling corresponds essentially to the dynamic scheduling performed by the 21264. Observe that two ingredients were necessary for this type of static scheduling to work. First, we assumed that the iterations are independent, which we may not infer from the code alone, since \( y_k \) and \( x_{k+1} \) may refer to the same memory location, for instance. Second, we require more registers for the statically scheduled code, whereas the dynamically scheduled code has its registers renamed at runtime.

The following list describes some advantages of using dynamic scheduling and shows ways to achieve similar efficiency by using static scheduling.

(i) The order of loads from non-cache memory in the instruction stream is not very important for dynamically scheduled processors. The delays for different loads within the instruction window—whose length is determined by the size of the issue queues—will overlap in any case. We have seen above how to reach this goal by reordering instructions for static scheduling. Another option is to use preloading or prefetching instructions to load the desired values into the cache as follows:

\[
\begin{align*}
    \text{daxpy}_{k...k+1} \equiv \\
    \langle x, x' \rangle &= \langle x_k, x_{k+1} \rangle \\
    \langle y, y' \rangle &= \langle y_k, y_{k+1} \rangle \\
    \langle t, t' \rangle &= \langle \alpha x, \alpha x' \rangle \\
    \langle t, t' \rangle &= \langle t, t' \rangle + \langle y, y' \rangle \\
    \langle y_k, y_{k+1} \rangle &= \langle t, t' \rangle
\end{align*}
\]

The particular order of the loads in the instruction stream does not matter that much if the values are already in the cache. Note further that preloading may even help to avoid introducing additional variables.

(ii) If the scheduling is done at runtime, we have more information than at compile time. In the example above it was not possible to know that \( y_k \) and \( x_{k+1} \) do not refer to the same memory location at compile time, while this is easy to find out at runtime. This is undoubtedly an important advantage, but on the other hand the programmer often knows that arrays like \( x \) and \( y \) are disjoint and that the corresponding loads and stored are interchangeable. For simple problems it is even feasible to check for disjoint arrays at runtime and select between different codes for the general case and the disjoint case.
(iii) In a dynamically scheduled processor, it is often possible to determine branch targets early, that is while the issue queues are still able to deliver instructions to the units. In optimal cases, there will be no branch penalty involved independent of the correctness of the branch predictor, or the cost involved in mispredicting can at least be significantly reduced. While it is not usually implemented, it is conceptually simple to do something similar with statically scheduled code. For a sample daxpy code, we may write symbolically

\[
\text{daxpy} \equiv
\]

\[
\text{loop}:
\]

\[
(x, y) = (x_k, y_k)
\]

\[
k = k + 1
\]

\[
t = ax
\]

\[
\text{if } k < n \text{ jump endloop} \rightarrow \text{loop}
\]

\[
t = t + y
\]

\[
y_{k-1} = t
\]

\[
\text{endloop}:
\]

This way, the instruction fetch unit can be notified of a control flow change before the actual branch happens. A special case of this idea was available for early RISC processors in the form of delay slots, which still deserve some merit. The technique of computing the branch target early tries to avoid fetching too many instructions from the wrong place. The complementary approach speculative execution is implemented in the 21264 as well, and will be discussed later.

The common theme of the points above is the question how to achieve the same scheduling with static scheduling as with dynamic scheduling. There are different obstacles to achieve this goal.

(i) The compiler does not usually know if a particular value is in the cache or not, so the result delay is undetermined at compile time. At runtime however, the delay will be visible, and dynamic scheduling allows to schedule instructions later, if they happen to depend on a delayed load. We see therefore that one important piece of information is the result delay of instructions, that is the number of cycles until the result is available.

(ii) Data dependences are known exactly at runtime, by simply looking at the memory addresses. At compile time, two different pointers may or may not point to the same memory location, and usually the compiler has to be conservative and assume that there might be a data dependence, although the programmer often knew there is not. Therefore, another piece of information is complete knowledge about data dependences.

(iii) Instructions responsible for the control flow like branches may not be moved freely since they carry the information about the starting point of the branch implicitly with their location in the instruction stream. So, the particular representation of the instruction stream may limit the ability to schedule instructions freely.

Let us assume that the delays of all instructions are known, that we have complete information about data dependences, and that there are no artificial restrictions which limit static instruction scheduling. Assuming further that the dependence structure within loops is identical for each instance of the loop during execution time, then static scheduling will be nearly equivalent to dynamic scheduling, given identical resources like registers and functional units.
**F.3 Speculative execution**

Speculative execution is another feature which is implemented in the 21264 but not in the 21164, and which also hurts the clock rate. The idea of speculative execution is to execute instructions which are control dependent on a not yet evaluated condition. Since it may turn out that these instructions should not have been executed at all, there are a couple of complications with this approach, since no side effects like generating exceptions or writing to memory permanently may be tolerated. For this reason, only particular types of instructions may be executed speculatively, and possible exceptions are not signaled until it is clear that the corresponding instruction required being executed. This type of administration is usually done by a retirement unit which commits instructions in order, although they may be issued and executed out of order.

Speculative execution is most profitable if the processor would lie idle while determining the control flow, and if the guessed control flow turned out to be the correct one. Like with dynamic scheduling, the usefulness of speculative execution is debatable. Of course, given the choice between speculative execution or not and everything else being the same, speculative execution is better. However, the true choice is between speculative execution and no speculative execution, but with a simpler—and therefore smaller and faster—processor. Speculative execution in hardware is unnecessary if we may compute branch targets early or if we cannot guess the correct control flow often enough. In some cases, speculative execution can be done in software as well. All we can say is that hardware based speculative execution is not a good choice for our MMM application. Your mileage may vary depending on the type of applications and your willingness to perform manual optimizations, since compilers are usually not allowed to schedule instructions for speculative execution.

An illustrative example of speculative execution in software is the computation of pair lists in the context of computing a pairwise potential. The problem we want to solve is the following. Given two sets of particle coordinates \( a_1 \ldots a_m \) and \( b_1 \ldots b_n \), determine the pairs \((a_i, b_k)\) for \( 1 \leq i \leq m \) and \( 1 \leq k \leq n \) such that the distance between \( a_i \) and \( b_k \) is smaller than a given cutoff radius, that is \( \|a_i - b_k\| < r = r_{cut} \). Note that computing the square of the distance suffices to perform the comparison, so a straightforward algorithm might look

```c
function pairlist  
    for i = 1...m do  
        \((x_0, y_0, z_0) = a_i\),  
        for k = 1...n do  
            \((x_b, y_b, z_b) = b_k\),  
            \(\rho^2 = (x_0 - x_b)^2 + (y_0 - y_b)^2 + (z_0 - z_b)^2\),  
            if \(\rho^2 < r^2\) then  
                append \((i, k)\) to pairlist  
                save \(\rho^2\)  
            fi  
        od  
    od
```

To understand the ideas involved it is necessary to switch to the corresponding representation in C code, where we may restrict our attention to the `if` statement.
Speculative execution

...  
if (rho2 < r2) {  
  *pairlist = k;  
  pairlist++;  
  *rho2list = rho2;  
  rho2list++;  
}  
...

where we have chosen a suitable representation of the pairlist. Now the speculative part consists in executing the stores unconditionally. This case illustrates that software based speculative execution may be even more flexible than hardware based speculation, which has to avoid any side effects. The code then looks

...  
*pairlist = k;  
*rho2list = rho2;  
if (rho2 < r2) {  
  pairlist++;  
  rho2list++;  
}  
...

We would already have gained something if we could exploit the early computation of the branch target. In this particular case, we may get rid of the branch altogether by simply computing the conditional, that is

...  
inc = (rho2 < r2);  
*pairlist = k;  
*rho2list = rho2;  
pairlist += inc;  
rho2list += inc;  
...

assuming that the result of a conditional is either 0 or 1. Although this is formally correct, it is still an oversimplification, so if you want to know the real truth, read on. The Alpha 21164 has no direct data path from the floating point units to the integer register file, so the result of a comparison between floating point values ends up in a floating point register. Since we need the result of the comparison in an integer register, and we have to make sure it is either 0 or 1, we check for the sign bit of the floating point number $r^2 - \rho^2$. The true code to compute inc would look

...  
d_dist = r2 - rho2;  
i_dist = *(int *)&d_dist; /* move msb(double) --> int bitwise */  
inc = (i_dist > 0);  
...

where it suffices to extract the most significant bits from the double.
It is debatable if this particular optimization may be called speculative execution. The reason why we think it qualifies for this token is that stores may be executed unnecessarily. The fact that we end up with a piece of code without any branch may not fit into the usual pattern of speculative execution, however.

F.4 Preloading

While processors are getting faster and more powerful at an amazing rate, memory access times are only slowly decreasing. This divergence of speeds can be coped with by using small and relatively fast expensive caches and large and relatively slow memories, making use of data access locality. But even with several levels of caches present, many applications are fundamentally limited by the bandwidth between the slow memory and the processor. Since increasing bandwidth is possible even if the technology does not allow for improved access times, high performance computer systems have to offer high memory bandwidths to the processor. However, the processor can only make use of this high bandwidth by appropriately preloading data from memory, therefore hiding the relatively large load latency.

Current microprocessor architectures define optional prefetching instructions, but an implementation only makes sense if the required bandwidth is available. On the 21164, there are two different ways to prefetch or preload values into the cache. First, there is a special instruction \texttt{fetch} which serves as a hint that a significant part of the data within a particular block may be used soon. If the system implements prefetching at all, the size of the block is expected to be about 512 bytes. At most two prefetches may be outstanding, and the data will be loaded into the on-chip second-level cache of the 21164. Second, a load into the special zero register \texttt{r31} preloads cache lines into the first-level cache. It is therefore possible to load or preload data in three stages in the Alpha architecture. First by using \texttt{fetch} to move large blocks from memory to the second-level cache, then by using loads into \texttt{r31} to move cache lines into the first-level cache, and finally the usual load instruction to load a few bytes from the cache into a register.

The usage of \texttt{fetch} only make sense if there is enough bandwidth, which is not the case for the machine containing the 21164 we used for the experiments. However, we expect that most systems containing the 21264 will offer enough bandwidth, at least up to memory requirements of a few megabytes, simply because this is one of the main differences to the 21164. Prefetching will then make sense and—if correctly implemented—might have a dramatic impact on the performance of codes which are bandwidth limited right now.

The big problem we have to face is that software cannot rely on the simple principle of locality alone, but it has to explicitly prefetch data. In simple cases, prefetching instructions can be inserted by the compiler. A typical case is given by a loop-unrolled version of \texttt{daxpy}, which looks as follows after compilation

\begin{verbatim}
function daxpy \equiv
  for k = 0, 4, 8 \ldots n - 4 do
    cache \{x_{k+8} \ldots x_{k+11}\} --- preload
    cache \{y_{k+8} \ldots y_{k+11}\} --- preload
    \langle t_0 \ldots t_3 \rangle = \langle \alpha x_k \ldots \alpha x_{k+3} \rangle --- 4 \Phi, 4 \Theta
    \langle t_0 \ldots t_3 \rangle = \langle t_0 \ldots t_3 \rangle + \langle y_k \ldots y_{k+3} \rangle --- 4 \Phi, 4 \Theta
    \langle y_k \ldots y_{k+3} \rangle = \langle t_0 \ldots t_3 \rangle --- 4 \Phi
  od
\end{verbatim}
This type of preloading has the following disadvantages. First, it is only efficient for relatively long vectors, since the first few elements are not preloaded. Furthermore, more elements than necessary are preloaded, which may be a problem or not depending on the available bandwidth. Second, the distance between prefetching and loading is limited by the number of outstanding loads the processor may cope with. In the Alpha 21164, pending loads are merged into pending cache lines, but at some point pending cache lines will fill the load buffer as well. Depending on the required bandwidth, the available cache bandwidth, and the given memory latency, this scheme may be reasonably efficient. Given a fast processor, a piece of code with high memory requirements, and high bandwidth, it is likely that the latency can only be hidden by using explicit preloading instructions which do not fill the load buffer.

Given a relatively low bandwidth between processor and memory, the above scheme will suffice to obtain optimal performance within the limitations imposed by the bandwidth. In more general cases, it is interesting to consider a few alternatives of preloading schemes. For illustration purposes we look at a few examples.

(i) In daxpy the access of the vectors $x$ and $y$ happens in a strictly sequential way. The sequence of memory accesses for a single vector is illustrated by the following graphic

![Graphic of memory accesses for daxpy](image)

Assuming that the vector length is not too small, it makes sense to preload parts of the vector which will be needed later. The sequence of memory accesses and preloads for a single vector then looks

![Graphic of memory accesses and preloads for daxpy](image)

As we have seen, the compiler is able to insert appropriate preload instructions for this type of sequential access. Instead of looking at the sequence of memory accesses, it is often sufficient to consider only the preloaded and loaded parts of the memory as shown below

![Graphic of preloaded and loaded parts of memory](image)

(ii) In phases $\bar{q} \rightarrow \bar{\sigma}$ and $\bar{\Sigma} \rightarrow \phi$ the structure of memory accesses is not purely sequential. The data for $\bar{\sigma}$ and $\bar{\Sigma}$ is contained in blocks which are not necessarily adjacent in memory. Using a simple sequential preloading scheme, we have the following situation
Although sequential preloading does improve performance in this case as well, the effect is like working with relatively short vectors and therefore having a significant part of the data not being preloaded and the same amount of memory being preloaded unnecessarily. For the programmer it is easy to apply a more efficient preloading scheme if this block structure is known. A possible sequence of blocked loads and preloads might look

While this scheme is not particularly complicated, it may be even simpler to preload a complete block while working on the previous one. In cases where the blocks are of equal size, this simplifies the placement of preloading instructions considerably. The sequence of loads and preloads would then look like

Note that even a simple scheme like this is not usually detected by the compiler, since information like equal block sizes or the fact that such blocks are handled in succession is usually not accessible to the compiler, or at least very hard to find out. For the programmer, however, these contexts are often obvious from the design of the program. For the phase \( \sigma \rightarrow \Sigma \) we may think of similar preloading schemes, with the only difference that we do not have contiguous blocks as units but some data structure which is spread in a regular way.

(iii) Although this is rarely the case in our particular application, irregular memory accesses may occur in more general cases. One type of irregularity is when the algorithm changes from one stage to the next, for instance. In our application, this might be the transition from phase \( \sigma \rightarrow \Sigma \) to phase \( \Sigma \rightarrow \phi \). For the illustration we will assume that the blocks accessed are all of the same size, so that irregular memory access might look like

If we do not mind preloading values which are already in the cache, then preloading becomes trivial, at least in the graphics
There are several reasons why preloading is not so simple in the actual code. First, the block sizes are usually not equal and only known at run time. Second, the loads may belong to different procedures or even different parts of the algorithm, so any knowledge of future memory accesses may be difficult to obtain, it may even be undesirable to exploit this piece of information from a software management point of view. Third, control dependences may prevent preloading at all, in other words the algorithm may not know early enough which part of the memory will be used later. No doubt, many other obstacles prevent an easy solution of the preloading problem in general cases.

We have seen that preloading instructions may be inserted by the compiler for simple codes like daxpy. For regular problems, the programmer is often able to implement simple and efficient preloading schemes. For irregular problems, the same task may be tedious, challenging, or even unreasonably hard. Preloading in general would be simplified enormously if we could have separate instruction streams for normal code and its \( \pi \) counterpart, whose purpose is to preload data from memory to the cache. For our daxpy example, the preloading piece would look

\[
\text{function daxpy}_\pi \equiv \\
\text{for } k = 0, 4, 8 \ldots n - 4 \text{ do} \\
\text{cache } \langle x_k \ldots x_{k+3} \rangle \\
\text{cache } \langle y_k \ldots y_{k+3} \rangle \\
\text{od}
\]

and the regular code would not require any preloading instructions. Given the regular instruction stream and the \( \pi \) instruction stream, the processor would make sure that the \( \pi \) stream is always ahead of the regular stream by a margin which covers the load latency from memory. This type of support is not required for regularly structured memory accesses, since the programmer is able to combine the regular and \( \pi \) streams at compile time for these problems.

F.5 Conclusion

We emphasize that the conclusions we may draw from the analysis of our MMM application are only valid for this particular application running on a particular machine with an Alpha 21164 microprocessor. In particular, we do not claim that features like dynamic instruction scheduling or speculative execution are useless for all applications. However, we think that the implementation of efficient preloading schemes might make some of these advanced features less important. We have listed a few possible consequences of this development below.

Manual optimization. Programming nowadays is done almost exclusively using high-level languages like Fortran and C, except maybe for a few high performance numeric kernel codes. This is justified to some extent thanks to optimizing compilers, which achieve good results while keeping the cost of software construction reasonably low. However,
in some cases manual optimization is still useful, although this does not necessarily mean programming in assembly. It is possible to write C or even Fortran code such that one high-level instruction corresponds to one assembly instruction or at least to a well-defined sequence of assembly instructions. Together with a compiler like gcc which allows a flexible selection of optimizations it should perform and which not, this is much more comfortable than assembly programming. We have seen many examples in the text which illustrate that instruction scheduling in general, software pipelining in particular and even register assignment being done manually may significantly improve the runtime of relatively simple routines.

**Dynamic scheduling.** We think that preloading will be one of the most important features of future microprocessor, or more correctly of computer systems including the processor-memory interface. Once preloading is common, one important reason for dynamic instruction scheduling, namely issuing loads as early as possible so that delays overlap or may even be partially hidden, will be gone. This does not imply that future processors will not offer dynamic scheduling, but we speculate that the price for implementing dynamic scheduling in terms of reduced clock rates is currently too high.

**Speculative execution.** Just like it is possible to preload data from memory, it is possible to some extent to determine branch targets ahead in time. The number of times a loop will be executed is often known right from the beginning, and some conditions for branches can be evaluated early. Numeric codes in particular usually spend most of their time in loops, so complicated branch prediction schemes and speculative execution would not be strictly necessary if there was a way to exploit the fact that branch targets may be computed early.

**Preloading.** Unless some breakthrough in technology will allow memory access times to speed up equally fast as the processors themselves, the technique of preloading will play the most prominent role for some time to come. Current architectures already support simple types of preloading, and it is an open question if anything much more complicated than explicit preload instructions can be effective. However, it is important to note that preloading is not something we should ask the compiler to do except in simple cases. Manual intervention is particularly useful because the structure of memory accesses can often be seen clearly from the design of an algorithm, but it may look complicated in the implementation.
References


Curriculum Vitae

Person.

Name: Rolf Strebel
Date of birth: November 29, 1967
Place of birth: Chur, Switzerland
Citizenship: Chur, Switzerland

Education.

1981–1987  Kantonsschule Chur, Matura Typus C
1988–1992  Study in computer science (Informatik) at ETH Zürich
            Work at HIWARE writing compilers, . . . for embedded systems

Interests.

▷ Programming in general
▷ Numerical analysis, particularly eigenvalue problems
▷ Compiler construction, optimization
▷ Mathematics and theoretical physics in general
▷ Backgammon and juggling