NTNU
Innovation and Creativity

Parallelizing Particle-In-Cell Codes with OpenMP and MPI

Nils Magnus Larsgård

Master of Science in Computer Science
Submission date: May 2007
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

This thesis searches for the best configuration of OpenMP/MPI for optimal performance. We will run a parallel simulation-application on a modern supercomputer to measure the effects of different configurations of the mixed code. After analyzing the performance of different configurations from a hardware point of view, we will propose a general model for estimating overhead in OpenMP loops.

We will parallelize a physics simulation to do large scale simulations as efficient as possible.

We will look at typical physics simulation codes to parallelize and optimize them with OpenMP and MPI in a mixed mode. The goal of the parallelization is to make the code highly efficient for a parallel system and to make it as scalable as possible to do large-scale simulations as well as using this parallelized application as a benchmark for our MPI/OpenMP tests.

Assignment given: 20. January 2007
Supervisor: Anne Cathrine Elster, IDI

Abstract

Today's supercomputers often consists of clusters of SMP nodes. Both OpenMP and MPI are programming paradigms that can be used for parallelization of codes for such architectures.

OpenMP uses shared memory, and hence is viewed as a simpler programming paradigm than MPI that is primarily a distributed memory paradigm. However, the Open MP applications may not scale beyond one SMP node. On the other hand, if we only use MPI, we might introduce overhead in intra-node communication.

In this thesis we explore the trade-offs between using OpenMP, MPI and a mix of both paradigms for the same application. In particular, we look at a physics simulation and parallalize it with both OpenMP and MPI for large-scale simulations on modern supercomputers.

A parallel SOR solver with OpenMP and MPI is implemented and the effects of such hybrid code are measured. We also utilize the FFTW-library that includes both system-optimized serial implementations and a parallel OpenMP FFT implementation. These solvers are used to make our existing Particle-In-Cell codes be more scalable and compatible with current programming paradigms and supercomputer architectures.

We demonstrate that the overhead from communications in OpenMP loops on an SMP node is significant and increases with the number of CPUs participating in execution of the loop compared to equivalent MPI implementations. To analyze this result, we also present a simple model on how to estimate the overhead from communication in OpenMP loops.

Our results are both surprising and should be of great interest to a large class of parallel applications.

Acknowledgments

The greatest contribution to and inspiration for this work has been Dr. Anne C. Elster. Without her previous works and her well documented code, my understanding of advanced physics and fourier solvers would have been close to nothing. Her famous brownies every Friday has also made the High Performance Computing(HPC) Group at NTNU to a great place to make friends and exchange opinions and experiences in the HPC field.

Thanks to Jan Christian Meyer for giving me his PIC codes and a good introduction to the field of particle simulation.

Thanks to all my friends and peers in the HPC group and the people at the computer-lab “Ugle” for ideas and feedback on coding concepts and theories.

I would also give a huge thanks to the people maintaining the Njord supercomputer at NTNU. I have not had any problems related to neither software or hardware on Njord during this project.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Programming for parallel computing has been dominated by the MPI and OpenMP programming paradigms. Both of the paradims aims to provide an interface for high performance, but their approach is somewhat different.

OpenMP is designed to shared memory systems, and has recently gained popularity because of its simple interface. With little effort, loops can easily be parallelized with OpenMP, and much of the synchronization and data sharing is hidden from the user.

MPI is designed for distributed memory and is probably the best known paradigm in parallel computing. Communication between processes is done explicitly, and a relatively large set of functions in the opens up for high performance and tweaking which is not available in OpenMP. Even though it is designed for distributed memory systems, it runs just as good on shared memory systems.

Computer systems with both of OpenMP and MPI available opens up for a hybrid programming style where both are used in the same application. The challenge on such systems is to find the optimal combination of the two programming styles to achieve the best performance.

1.1.1 Thesis Goal

This thesis searches for the best configuration of OpenMP/MPI for optimal performance. We will run a parallel simulation-application on a modern supercomputer to measure the effects of different configurations of the mixed code. After analyzing the performance of different configurations from a hardware point of view, we will propose a general model for estimating overhead in OpenMP loops.

We will parallelize a physics simulation to do large scale simulations as efficient as possible.

We will look at typical physics simulation codes to parallelize and optimize them with OpenMP and MPI in a mixed mode. The goal of the parallelization is to make the code highly efficient for a parallel system and to make it as scalable as possible to do large-scale simulations as well as using this parallelized application as a benchmark for our MPI/OpenMP tests.

1.2 Terminology

- **PIC codes** - Particle-In-Cell codes, a popular particle simulation technique for collision-less charged particles in electromagnetic fields.
- **Processors** - A physical processing unit. Several processors can be on one physical chip; dual-core consists of two processors on one chip.
- **Processes** - A program with private memory running on one or more processors. A process can spawn several child processes or threads to utilize several processors.
- **Threads** - Threads are lightweight processes that can have both shared and private memory. Processes consists often of two or more threads.
- **Nodes** - One or more interconnected processors with shared memory. The interconnection does not consist of regular network, but of high-speed mediums as a fixed databus on a motherboard.
- **FFT** - Fast Fourier Transform. Used as a direct solver for partial differential equations.
- **SOR** - Successive Over Relaxation. Used to obtain an approximation of a solution for partial differential equations.

1.3 Thesis Outline

Chapter 1 contains this introduction.

Chapter 2 summarizes some of the important background theories and related work for this thesis. A short introduction to OpenMP and MPI is given, along with some de facto models for performance modeling. Classifications of supercomputer architectures are presented. The PIC simulation is also presented with some references to where more information can be found.

Chapter 3 describes the two PIC simulation codes we work on, and some general background on how the general algorithm of PIC simulations are. Important steps of the algorithm is described and the SOR and fourier solver are described. Both Jan Christian Meyer's[3] and Anne C. Elster's simulation[1] are described, with some comments on initial performance and potential performance improvements.

Chapter 4 contains a summary of the main changes done to the simulation codes. The main changes are that two new solvers are introduced in Elster's simulation code in addition to dealing with parallel issues of other parts of the program.

Chapter 5 presents the performance results from our tests for both the Meyer and Elster versions of the PIC simulation. Different configurations of threads and processes on the SMP nodes are tested and the results are presented. We also give a detailed analysis of the performance results. We also suggest a model for estimating the overhead when using OpenMP on more than one processor chip. The model is highly connected with the hardware performance, more specifically memory latency and memory bandwidth.

Chapter 6 concludes the thesis and present some future works as a continuation of this thesis.

Chapter 2

Background Theory

A dwarf on a giant's shoulders
sees farther of the two

Jacula Prudentum

This chapter gives a short introduction to the theory and technology used in this project.

Some general introduction to performance modeling and Amdahl's laws are given in Section 2.1. The MPI and OpenMP API's are introduced in Section 2.2 and 2.3. Programming models with these two API's are described in Section 2.4. Some common memory architectures are described in Section 2.5 along with what programming models are suited for each of them. A short introduction to PIC-codes are given in Section 2.6. The math libraries used in this project are described in 2.8.

Previous and related works are described in Section 2.10, including the PIC codes this thesis is built upon and other existing PIC codes. An article on hybrid parallel programming is summarized in 2.10.1.

2.1 Performance Modeling

The Laws of Amdahl are very general laws about performance and efficiency of parallel applications. These laws are expressed by formulas for how well parallel programs scale and how efficient they are compared to serial versions of the same program, and are therefore often referred to when modeling performance for MPI.

2.1.1 Amdahl's Law

Amdahl's Law[12] describes maximum expected improvement for an parallel system and is used to predict the maximum speedup of a parallel application. If a sequential program executes in a given time T_σ and the fraction r of the program is the part which can be parallelized, the maximum speedup $S(p)$ for p processes is found by the formula

$$S(p) = \frac{T_\sigma}{(1-r)T_\sigma + rT_\sigma/p} = \frac{1}{(1-r) + r/p} \tag{2.1}$$

If we differentiate S with respect to p and let $p \to \infty$, we get

$$S(p) \to \frac{1}{1-r} \tag{2.2}$$

from [12].

This implies a theoretical limit of the speedup of a program when we know the fraction of parallel code in the program.

2.1.2 Work and Overhead

Even if we parallelize the code perfectly, a parallel program will most likely have a worse speedup than predicted with the Amdahl's Law. This is mainly due to overhead from the parallelization. The sources of overhead are many, the most known are communication, creation of new processes and threads, extra computation and idle time.

The total work done by a serial program W_q is the runtime $T_\sigma(n)$. The work done by a parallel program W_π is the sum of the work done by the p processes involved(2.3). Work includes idle time and all the extra overhead in the parallelization.

$$W_\pi(n) = \sum W_q(n,p) = pT_\pi(n,p) \tag{2.3}$$

from [12].

Overhead T_O is defined as the difference in work done by a parallel implementation and a serial implementation of a given program. The formula is given in equation 2.4. Overhead is mainly due to communication and the cost of extra computation when parallelizing an application.

$$T_O(n,p) = W_\pi(n,p) - W_\sigma(n) = pT_\pi(n,p) - T_\sigma(n) \tag{2.4}$$

from [12].

2.1.3 Efficiency

From the definitions of work and overhead we can derive the efficiency as the work done by the serial application compared to the work done by the parallel application. If we have no overhead at all(the ideal parallel application), the efficiency is 1. To achieve 100% efficiency, no same thing should be computed on more than one node. The formula for efficiency is given in equation 2.5.

$$E(n,p) = \frac{T_\sigma(n)}{pT_\pi(n,p)} = \frac{W_0(n)}{W_\pi(n,p)} \tag{2.5}$$

from [12].

In theory, efficiency above 100% is impossible, but is in practice achieved because of large cache sizes or the nature of datasets. Speedup above 100% for parallel applications is called superlinear speedup.

Communication

Communication is a source of overhead when parallelizing programs. The time to communicate a n-size message is roughly expressed with latency, T_s, and bandwidth, β, as

$$T_{comm} = T_s + \beta n \tag{2.6}$$

.

2.2 The Message Passing Interface - MPI

MPI[16] [24] is an industry standard for message passing communication for applications running on both shared and distributed memory systems. MPI allows the programmer to manage communication between processes on distributed memory systems. The MPI is an interface standard for what an MPI-implementation should provide of functions and what these functions should do. There are several implementations of MPI and the best known are probably MPICH and OpenMPI, both open source version implementation. Most vendors of HPC resources also have their own proprietary implementation of MPI with bindings for C, C++ and Fortran. SCALI[4] is probably the best known Norwegian vendor of MPI implementations.

The first MPI standard was presented at Supercomputing 1994[1] and finalized soon thereafter. The first standard included a language independent specification in addition to specifications for ANSI-C and Fortran-77.

About 128 functions are included in the MPI 1.2 specification. This interface provides functions for the programmer to distribute data, synchronize processes and create virtual topologies for communication between processes.

2.3 OpenMP

Shared memory architectures open up for efficient use of threads and shared memory programming models. The traditional way to take advantage

[1]November 1994

of such architectures is to use threads in some way or other. POSIX threads(pthreads) are most used in HPC programming, however thread models can quickly generate complex and unreadable code.

Recent advances in processor architectures with several cores on the same chip has made shared memory programming models more interesting. Especially the introduction of dual-core processors for desktops the last two years[2] has made this area of research interesting for other groups than the HPC communities.

OpenMP is an API that supports an easy-to-use shared memory programming model. The easiness of inserting OpenMP directives into the parallel code has made this model popular compared to pthreads. This model leaves most of the work of thread handling to the compiler and greatly reduces the complexity of the code.

The directives for parallelization in OpenMP allows the user to decide what variables that should be shared and private in an easy way, in addition to what parts of the code that should be parallelized. The simplicity of using OpenMP directives has made it a popular way of parallelizing applications.

2.4 Hybrid Programming Models

2.4.1 Classifications

In [14] Rolf Rabenseifner describes a classification scheme over hybrid programming models, based on if they use OpenMP, MPI and mixed versions of the two programming paradigms.

1. **Pure MPI:** Only MPI is used, and each processor on the target system runs an MPI process with its own memory. The MPI library must take care of intra-communication in a node and intercommunication between nodes. This is well suited for distributed memory models, and does not take advantage of multicore processors or SMP nodes.

2. **Pure OpenMP:** Only OpenMP is used to take advantage of SMP systems. This model is not suitable for systems that are distributed memory systems or connected shared memory systems.

3. **MPI + OpenMP without overlap:** This model is taking advantage of SMP nodes, and does inter-node communication with MPI.

   - **Master only:** MPI-operations are used only outside parallel regions of the code, by the master thread. OpenMP is used on each SMP-node.

[2]AMD released the first AMD-X2 in may 2005 and Intel released the Core Duo in January 2006, both architectures were targeted for main-stream consumers.

- **Multiple:** MPI-operations are used only outside parallel regions, but several cpus can participate in the communication. OpenMP is used on each SMP-node.

4. **MPI + OpenMP with overlapping communication and computation:** This is the optimal scheme for systems with connected SMP nodes. While one or more processors are handling communication, the other processors are not left idle but does useful computation.

   - Hybrid funneled: Only the master thread does MPI routines to handle communication between SMP nodes. The other threads can be used to do computation.
   - Hybrid multiple: Several threads can call MPI-routines to handle their own communication, or the communication is handled by a group of threads.

2.5 Supercomputer Memory Models

Supercomputers come in many forms and are in this project roughly divided into four groups: *distributed memory systems, shared memory systems, hybrid systems and grids.* The respective groups are described and illustrated with simplified figures below.

- Distributed memory systems(Fig. 2.1) share interconnection but have private processor and memory. These systems are well suited for message-passing libraries like MPI.

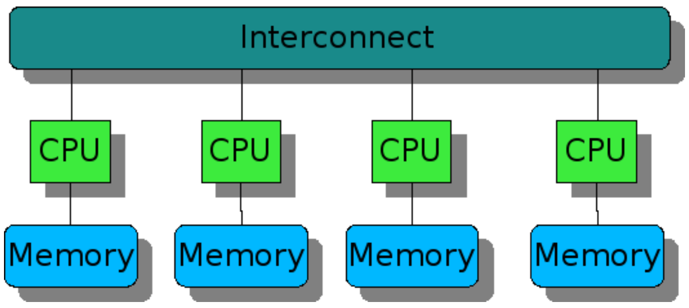

Figure 2.1: A distributed memory system.

- Shared memory systems(Fig. 2.2) are systems with more than one processor where all processor's share memory. These systems are well suited for OpenMP, MPI, or mixed OpenMP-MPI programming models. Each processor sees the memory as one large memory.

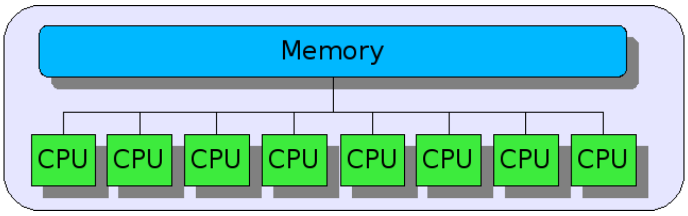

Figure 2.2: A shared memory system.

- Hybrid Systems(Fig. 2.3) consist of two or more homogeneous interconnected shared memory systems. These systems can run OpenMP and MPI on each node and use MPI for communication between the nodes. This is the architecture of the Njord supercomputer at NTNU.

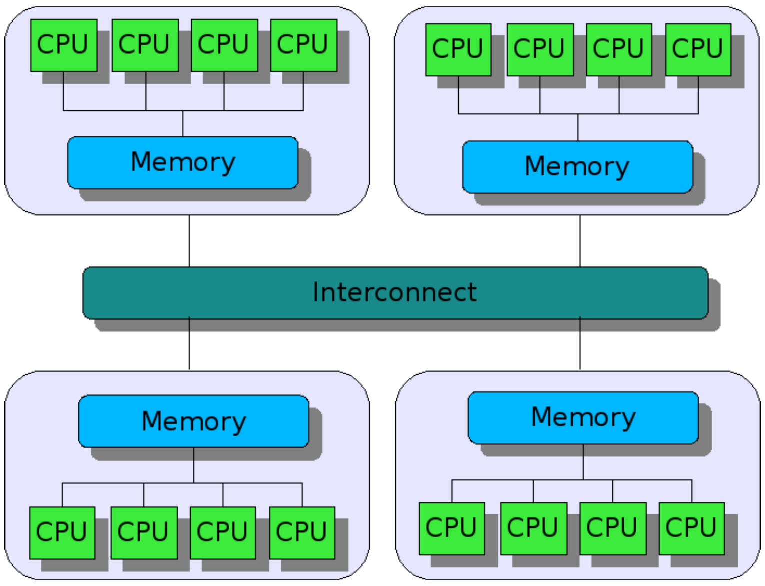

Figure 2.3: Connected shared memory nodes.

- Grids(Fig. 2.4) are heterogeneous systems where the only thing the components must have in common is the software. OpenMP can be deployed on the shared memory systems, and MPI can be used for communication between the nodes. Programming for grids can be more challenging than programming for the other models because of the heterogeneous architecture.

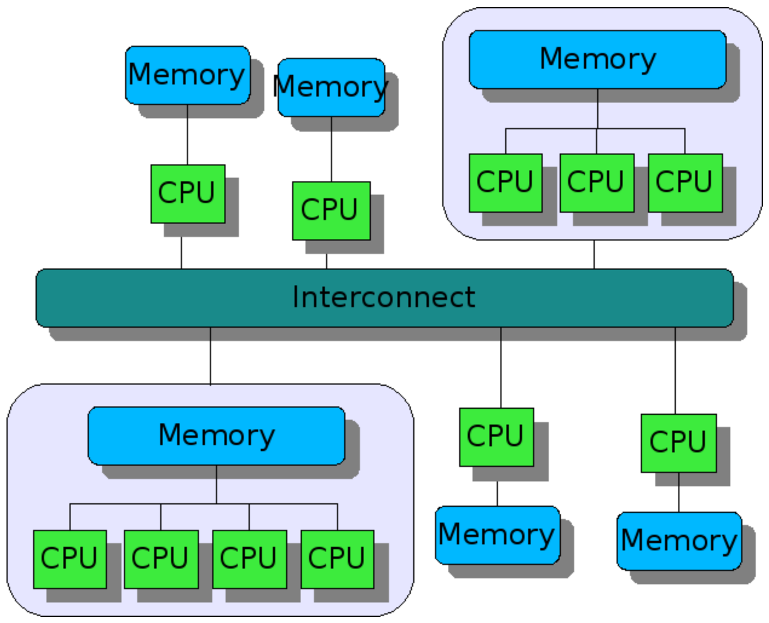

Figure 2.4: A Heterogeneous network/grid of multicore shared memory systems and single-core systems.

2.6 Particle-In-Cell Simulations

Particle-In-Cell(PIC) is a numerical approach to simulate how independent charged particles interact with each other in a electric or magnetic field. Such simulations are used in various research areas such as astrophysics, plasma physics and semiconductor device physics[1]. The parallelization of the PIC code done by Anne C. Elster has been used by Xerox in their simulations[5] to simulate how ink can be controlled by electric fields in standard ink-printers.

Depending on the size of these simulations, they can require more computation power than normal desktops offers and supercomputers therefore frequently used to perform PIC simulations.

Simulation of PIC involves tracking the movement, speed and acceleration of particles in electric or magnetic fields. The particles are moving in a matrix of 'cells' and contribute to the cells' charge closest to them.

Solving the field is the operation to compute the charge-contribution from each particle to the field and update the field with correct values. Numerical methods for solving the field includes fourier transforms and approximation methods like SOR.

For an extensive introduction to PIC codes, see [1]. For a lighter introduction, see [3].

2.7 PDE Solvers

By *solvers* we mean mathematical methods for solving a partial differential equation(PDE). The PDE can, as in our case, describe physical fields with some defined forces working on the field. The properties of the field defines what solvers can be used. Typical properties of the field includes periodic or non-periodic borders, linearity and dimensionality. In general we can define two types of solvers for PDEs: *direct* and *iterative*.

2.7.1 Direct Solvers and the FFT

Direct solvers gives an exact solution for the PDE. Simple PDEs without complex borders or with periodic border conditions are typically solved with direct solvers.

The Fourier Transform and the Discrete Fourier Transform, [1]

The Fourier Transform is a linear operator that transforms a function f(x) to a function F(w). The operator is described in detail in [1], but we will give a short summary here.

If the function $f(x)$ describes a physical process as a function of time, then $H(\omega)$ is said to represent its frequency-spectrum and ω are measured in cycles per time unit x. In our case, the function $f(x)$ describes distance, and the ω therefore describes angular frequency.

The fourier transform of a function $f(x)$ is a function $F(\omega)$:

$$F(\omega) = \int_{x=-\infty}^{\infty} f(x)e^{-i\omega x}dx \tag{2.7}$$

and the inverse transform takes the function $F(\omega)$ back to $f(x)$:

$$f(x) = \frac{1}{2\pi}\int_{x=-\infty}^{\infty} F(\omega)e^{i\omega x}d\omega \tag{2.8}$$

It can be shown [1] that the second derivative of the original function would be the same as multiplying the transform with a constant,

$$\frac{d^2f}{d^2x} = \frac{1}{2\pi}\int_{x=-\infty}^{\infty} (-\omega^2)F(\omega)e^{i\omega x}d\omega \tag{2.9}$$

and therefore we can compute the second derivative of $f(x)$ just as fast as a scaled fourier transform.

Since a computer cannot calculate continuous functions, The Discrete Fourier Transform (DFT) is constructed by discretizing the continuous fourier transform. Instead of integrating in a continuous space, we use summations for a finite set of grid-points. A 1D transform of a discretized signal of size N is obtained by

$$F(f(x)) = F(\omega) \approx \sum_{n=0}^{N-1} f_n e^{i\omega n} \tag{2.10}$$

and is applied on a 2D signal by applying the 1D transformation both horizontally and vertically. Our PDE can be considered as a 2D signal, so solving our PDE with DFT will give an exact solution for the field.

Evaluating a 2D signal with the original DFT would take $O(N^2)$ arithmetic operations. With the more efficient Fast Fourier Transform(FFT), the number of arithmetic operations are reduced to $O(N \log(N))$.

Parallel implementations of the DFT and FFT are not discussed here, but we refer to [8] and [9] for further readings on this topic.

2.7.2 Iterative Solvers and the SOR

Complicated equations with complex borders are often solved with iterative solvers. These solvers gives an approximated solution of the field after a defined number of iterations or until an error rate is defined as acceptable. One of these solvers is the SOR solver[10] which we will shortly describe here.

Jacobi and SOR

A Jacobi solver can be applied to the following the 2 dimensional field, u with m columns and n rows, defined by the following equation and its border cases.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \begin{cases} u(x,1) = 1 & \text{for} \quad 0 \leq x \leq 1 \\ u(x,0) = -1 & \text{for} \quad 0 \leq x \leq 1 \\ u(0,y) = u(y,0) = -1 & \text{for} \quad 0 \leq y \leq 1 \end{cases} \tag{2.11}$$

Note that the borders in this case are just for example, they can have any value that are constants.

To solve this system with an iterative solver, we approximate the (n+1)th iteration of the solution by applying the Jacobi iteration

$$u_{i,j}^{n+1} = \frac{1}{4}(u_{i-1,j}^{n} + u_{i+1,j}^{n} + u_{i,j-1}^{n} + u_{i,j+1}^{n}) \tag{2.12}$$

for all $j = 1 \ldots m$ and $i = 1 \ldots n$. This iteration is applied until the error rate, the difference between $u_{i,j}^{n-}$ and $u_{i,j}^{n}$, is acceptable.

To speed up the convergence of Jacobi, there has been developed several methods. The most common one is Successive Over Relaxation(SOR). This method assumes that the solution is “further ahead”, and takes larger steps than Jacobi to converge faster. The SOR iteration is shown in Equatioin 2.13, where ω is the degree of over-relaxation, usually between 1.7 and 1.85 according to [1]. In addition, it also takes use of points already updated in step $n+1$.

$$u_{i,j}^{n+1} = \frac{1}{4}(u_{i-1,j}^{n+1} + u_{i+1,j}^{n} + u_{i,j-1}^{n+1} + u_{i,j+1}^{n}) + (1-\omega)u_{i,j}^{n} \tag{2.13}$$

Red-Black SOR

An optimization of SOR is the Red-Black SOR scheme. In the original SOR, each point is dependent of the previous updated point, making it hard to parallelize.

By dividing the points into red and black points formed like a chess-board, one can easily see that red points only depend on black points and vice versa. Therefore we can perform iterations where every other iteration computes the red and black points as shown in Figure 2.5[3].

In parallel implementations with n nodes of the SOR solver for fields with size N, each node typically has a matrix of size $\frac{N}{n}$ in addition to ghost elements. Ghost elements are points received from neighbor nodes.

One SOR or Jacobi iteration takes $O(N)$ operations per iteration if the equation is N big. How many iterations needed depends on the defined maximum error rate.

For a further introduction to SOR solvers, see [10].

2.8 Math Libraries

When writing physics or math code, the use of ready libraries can ease the development and cut the development time drastically. Performance of many libraries are often significantly faster than home made code, so if they are used we can get substantial performance gains. In this project we intended to make use of the *FFTW*-libraries and the *PETSc*-libraries.

2.8.1 The "Fastest Fourier Transform in the West"

FFTW("The Fastest Fourier Transform in the West")[18], is a fairly new library which performs the discrete fast fourier-transform(fft). It freely available under the GPL license. It is written in C and is ported to most POSIX-compliant systems, and increasingly more popular library. The performance of the library can often compete with machine-optimized math libraries according to [18]. The library consists of several discrete fft-implementations called codelets. With normal usage of the library, several codelets are tested to find the codelet that performs best on the specific

[3]The figure is copied with permission from [6].

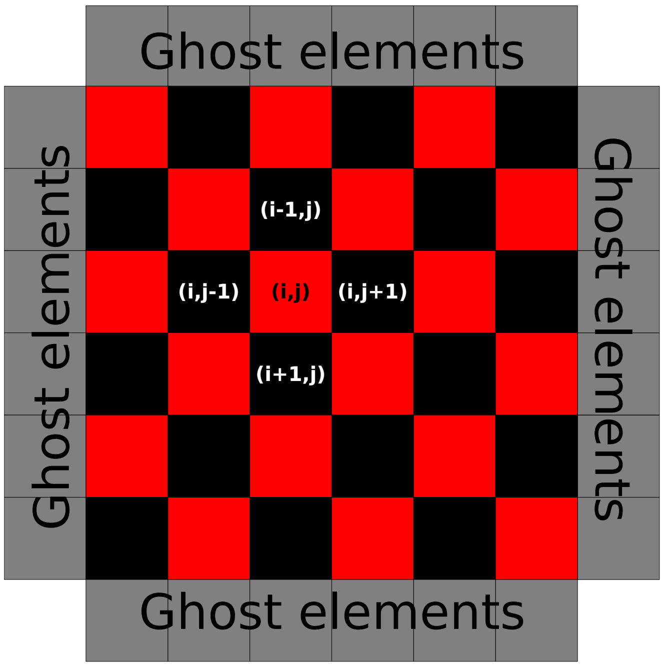

Figure 2.5: Red-black SOR with ghost elements. The red point (i, j) is only dependent on black points.

processor-architecture. This way the best implementation is used for any platform. For updated information on features of FFTW, benchmarks and documentation, see [18].

2.8.2 Portable, Extensible Toolkit for Scientific Computation

PETSc,(Portable, Extensible Toolkit for Scientific Computation)[19], is a parallel library that contains sequential and parallel implementation for solving partial differential equations. It uses MPI for it's parallelization, and lets the users interact with distributed vectors and matrices in a uniform way. It is well documented and contains a lot of examples.

2.9 Profilers

Profilers are programs which are made to give statistics about where time is spent in an application. This is very useful for locating hot-spots and bottlenecks in an application.

2.9.1 Software profiling: gprof, prof

The most used way to make a profile of an application, is to compile with the `-g -p` flags before it is run. This will reduce the performance of the application, but is only intended for use when optimizing or analyzing an application. Typical profilers are `gpof`, `prof` and `tprof`. For more information on these, see AIX 5L Performance Tools Handbook, Chapter 19 [11].

2.9.2 Hardware profiling: pmcount

To get a more hardware close profile of a program, program counters can be used. These programs provides statistics about the hardware performance from hardware counters, resource utilization statistics and derived metrics. For parallel programs running on Power5/AIX systems, `hpmcount` is widespread and easy to use. It requires very little effort from the user and gives useful statistics about the hardware utilization.

2.10 Related Work

This section gives a brief introduction to the most central references on which this project is built.

2.10.1 Hybrid Parallel Programming on HPC Platforms

Rolf Rabenseifner has done a study[14] of performance of hybrid parallel programming patterns, as well as defining different categories for hybrid models. In *Hybrid Parallel Programming on HPC Platforms* he evaluates performance on different architectures with the defined hybrid models. The goal is to find the hotspot-combination of MPI and OpenMP for performance on each platform.

2.10.2 Parallelization Issues and Particle-In-Cell Codes

Anne C. Elster wrote her PhD dissertation[1] at Cornell University in 1994 on *Parallelization Issues and Particle-In-Cell Codes.*

The dissertation is a comprehensive introduction to PIC simulation, and description of the field. Parallelization and memory issues are discussed in detail, along with a detailed description of the PIC algorithm.

The PIC-implementation from this dissertation uses a home-made FFT implementation and has option to use pthreads for the parallel version. Neither MPI or OpenMP were used.

The work described in this report is highly related to Elster's dissertation.

2.10.3 Emerging Technologies Project: Cluster Technologies. PIC codes: Eulerian data Partitioning

Jan Christian Meyer has written an other PIC implementation and a report on the performance of it on two different architectures. The report was written in connection with the Emerging Technologies Project and is called *PIC Codes: Eulerian data partitioning*[3].

The application uses MPI for parallelization, and a SOR solver.

At the very beginning of this project, Meyer's implementation was target for optimization.

2.10.4 UCLA Parallel PIC Framework

UPIC(The UCLA Parallel PIC Framework) [7] is a framework for development of new particle simulation applications and is similar to the previous PIC codes described. It provides several solvers for different simulations and has parallel support. It is written in Fortran95 and said to be "designed to hide the complexity of parallel processing" and has error checks and debugging helps included.

Since the authors of UPIC did not respond to our requests, we decided to build our own framework for highly parallel PIC codes.

Chapter 3

Particle-In-Cell Codes

All science is either physics or
stamp collecting.

Ernest Rutherford

In this chapter we give an introduction to the general theory behind PIC simulations and introduction to two different implementations. In section 3.1 the general PIC algorithm is described. Section 3.2 describes the PIC code implemented by Elster[1][2], while Section 3.3 describes the PIC code made by Jan Christian Meyer.

3.1 General Theory

The general algorithm for a PIC simulation in [1] is given in Figure 3.1, and the components of the algorithm as in [1] is described in the following subsections.

```
1: Initialize(field, particles)
2: while  t < $t_{max}$ do
     3: Calculate particles contribution to field
     4: Solve the field
     5: Update particle speed
     6: Update particle position
     7: Write plot files
```

Figure 3.1: An general algorithm of a PIC simulation

3.1.1 The Field and the Particles

The physical field is represented as a discrete 2D array with $Nx * Ny$ gridpoints. In the MPI-versions of PIC, each of the n MPI-processes is computing a $\frac{1}{n}$ share of the field.

The particles have 4 properties: charge, speed, location and weight. These values are described with double precision variables to give the needed amount of accuracy. Depending on the simulation, these properties have an initial value or zero-values.

The size of the grid describing the field determines how high resolution the simulation will have.

3.1.2 A Particles Contribution to the Field

Since the particles position is represented with double precision numbers, and the field is represented by a non-continuous grid, the particles seldom are located exactly on one grid point. With high probability, the particle is located as depicted in Figure 3.2 where the distance a and b varies. hy and hx are the grid-spaces in y and x direction.

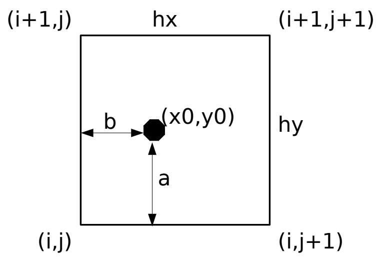

Figure 3.2: A particle is located in a grid-cell of the field, contributing to the charge of 4 grid-points.

Only the four grid-points surrounding a particle are updated, using the pseudo code in Listing 3.1 to update the grid-points. rho_k is a scaling factor computed from the particles charge and the grid-spaces.

Listing 3.1: Updating the charge field from a particles position.

```
field[i  ,j  ] += (hx - b) * (hy - a) * rho_k;
field[i,j+1] += b * (hy - a) * rho_k;
field[i+1,j] += (hx - b) *  a * rho_k;
field[i+1,j+1] += a * b * rho_k;
```

3.1.3 Solving the Field

The type of solver is chosen based on the properties of the field. Direct solvers like the fourier transform or iterative solvers like jacobi are typical solvers for the field, as described in Section 2.7.

3.1.4 Updating Speed and Position

To update a particles speed, we first need to find its acceleration. The acceleration is computed from the charge of the particle, q, and the forces from the field on the particle, E

$$F = q * E \tag{3.1}$$

where the field E can be decomposed into x and y direction, E_x and E_y.
Speed is simply updated for x and y direction with

$$v = v_0 + \frac{F}{m} \tag{3.2}$$

where m is the mass of a particle.

Updating a particles position is straight forward when we know its speed and current position, simply add the timestep,$\bigtriangledown t$, multiplied with the updated speed, v, in both x and y direction.

$$Pos_x = Pos_x + \bigtriangledown t * v_x \tag{3.3}$$

$$Pos_y = Pos_y + \bigtriangledown t * v_y \tag{3.4}$$

3.2 Anne C. Elster's Version

Anne C. Elster[1] implemented her PIC-application with an FFT solver for 2 dimensions. The implementation uses a hand-optimized FFT solver. Elster used an FFT algorithm based on "Numerical Recipes"[9] that she hand-optimized at that time. At the time the program was built(1994) optimized math-libraries were not as widespread as it is today. The program is parallelized with Pthreads to run on a shared memory architecture.

3.2.1 Program Flow

This was the program flow of Elster's original program.

1. Data describing the field and charges is read from input file `tst.dat`.
2. User types in size of grid to `stdin`.
3. Resources are allocated and speed and locations get initial values. Trace files are opened for a set of particles. Only particles are traced. Field charge is set to zero.
4. The field is updated with the charges of the particles.
5. Fourier solve of the field.

```
-1.6021773E-19
8.854187817E-12
9.109389E-31
-1.602E-12
0.0
0.000001 0.0001
1.0 1.0
```

Table 3.1: Sample input file for original program

6. The x and y components of the charge-fields are updated from the resulting field.
7. Speed is updated using the Leap-Frog model[1], and new position of the particles are found.
8. Simulation loop begins
   (a) Update speed and location of the particles
   (b) Reset field
   (c) Calculate particles charge-contribution to the field.
   (d) Solve the field.
   (e) The x and y components of the charge-fields are updated from the resulting field.
   (f) If simulation is not finished, go to 8
9. Print statistics of time usage, close up trace files.

3.2.2 An Example Run

Physical properties for the simulation were given in an input file as depicted in Table 3.1

3.2.3 Output, Timings and Plotdata

Plotdata were written to hdf-format, and timings of the various parts of the program is written to `stdout`.

3.2.4 Optimization Potentials

Since the program uses a home-made implementation of the FFT to solve the field, performance gains could be present if highly optimized libraries are deployed. Further parallelization with MPI and OpenMP can improve the scalability for the program, as well as make it possible to run it on non-shared-memory systems. The program is only tested for small grids, i.e. Nx and Ny below 256.

3.2.5 Initial Performance

A sample profile is shown in Table 3.2, and we can see that the numeric intensive parts of the code takes the most of the time. Optimization efforts can be put into the solver, `four1`, or the leap-frog method in `PUSH_V`. This profile is from running the program on a single cpu with 1 thread, Nx=128, Ny=128, number of particles=32.

Name	%Time	Seconds	Cumsecs	#Calls	msec/call
.four1	23.2	2.05	2.05	52224	0.0393
.PUSH_V	20.9	1.85	3.90	102	18.14
.floor	14.6	1.29	5.19	29743200	0.0000
.PART_RHO	14.0	1.24	6.43	102	12.16
.__mcount	6.4	0.57	7.00		
.FFT_SOLVE	5.9	0.52	7.52	102	5.10
.PUSH_LOC	5.1	0.45	7.97	101	4.46
._doprnt	4.3	0.38	8.35	437266	0.0009

Table 3.2: Initial output from `prof`, showing only methods consuming over 3.0% of total runtime on a single CPU.

3.3 Jan Christian Meyer's Version

Jan Christian Meyer[3] developed his code with inspiration from Anne C. Elster's code[1], but wrote the whole program over to consider N-dimensional simulations.

As a result of the option for N dimensions, the code suffers on the performance part. The SOR solver works correct, but the program is difficult to extend for other types of solvers.

3.3.1 Program Flow

The program flow is described in detail in [3], but we will give a short summary here:

1. Process 0 reads simulation data and broadcasts it to all processes.

2. Each process computes its own relation to the global grid, and allocates memory for its subgrid.

3. Process 0 reads particle number and the initial position and speed for each particle. The particles are sent to its respective positions and processes in the global grid.

4. Simulation loop begins:

    (a) All processes opens plot files for timings and particle data.

    (b) The charge is distributed from the particles in each subgrid of the field.

    (c) Each process sends the boundaries of their subgrid to its respective neighbour processes.

    (d) The field is solved both locally and globally, and field strength is obtained for each subgrid.

    (e) All subgrids interpolates the electric field strength at the particles current position.

    (f) Speed and position of particles are update. Particles are sent between processes if they cross subgrid borders.

    (g) If the simulation is not finished, go to 4.

5. All resources are freed.

3.3.2 An Example Run

This section will describe what is needed for execution of the program and output produced by the program.

Configuration and Inputs

The program is compiled and linked into one executable file, `simulation`. The executable takes two text-files as arguments, `input.txt` and `distribution.txt`. `input.txt` contains information about the field and the configuration of the solver as given in Table 3.3.

`distribution.txt` contains information about the particles in the simulation. The number of entries should be the same as specified in `input.txt`. Initial speed and location is given as in Table 3.4.

3.3.3 Output - stderr, Timings and Plotdata

The program has mainly three output types: `stderr`, information about timings and plotdata. `stderr` is used only to print the status of what iteration is finished.

The timings contains information for each step and for each process involved in the program on how long time is spent in the various regions of

```
9 particles
1e10 e-charges pr. particle
9.109389e-31 mass
0 drag

3e-13 time step
65 steps

2-dimensional grid
200x100 grid points
.005x.01 cell size
Boundaries:
            [ 0:1, 0:0]
               [1, 1]
```

Table 3.3: Sample `input.txt`

```
P(0.01 0.4960) V(0 0)
P(0.01 0.4970) V(0 0)
P(0.01 0.4980) V(0 0)
P(0.01 0.4990) V(0 0)
P(0.01 0.5000) V(0 0)
P(0.01 0.5010) V(0 0)
P(0.01 0.5020) V(0 0)
P(0.01 0.5030) V(0 0)
P(0.01 0.5040) V(0 0)
```

Table 3.4: Sample `distribution.txt`

```
0 spent 0.000038 distributing charge
0 spent 0.013229 transmitting charges
0 spent 9.978237 solving for the field
0 spent 0.016681 on I/O
0 spent 0.000001 displacing particles
0 spent 0.000134 migrating particles
```

Table 3.5: Sample timings of an iteration for process 0.

the program. This is very useful considering optimization of the program. A sample of timings is given in Table 3.5.

Plotdata is written to text-files in a format that is easily visualized by programs like `gnuplot`. The plotdata files contains information of the potential of the field, an example of visualization is given in figure 3.3.

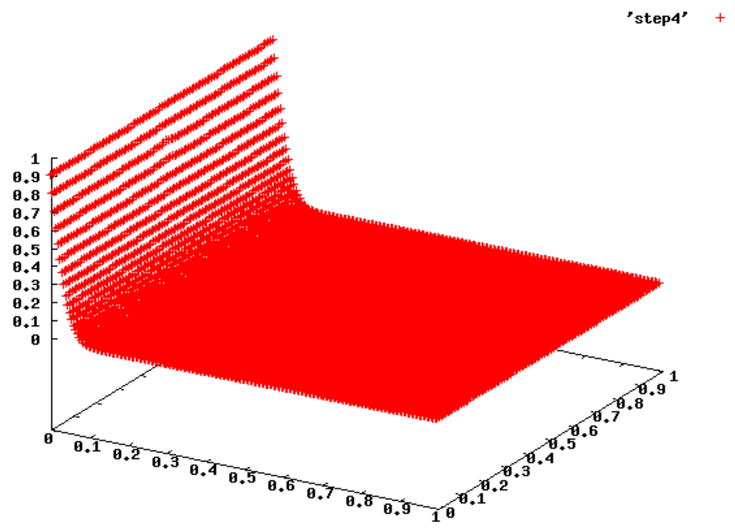

Figure 3.3: Sample visualization of potential in the field. Plotted in gnuplot from plotdata, step 4 of the simulation.

3.3.4 Optimization Potentials

The code was not written with optimal performance in mind [3], so the potential for improvements in runtime for the original code was indeed present. Since a simple and naive SOR-implementation is used, performance gains could be achieved by using a solver from a math library, e.g. an optimized fft-solver or an optimized SOR-solver.

3.3.5 Initial Performance

This subsection gives some numbers and statistics on performance of the original program.

Timings and Profiling

The output from the profiling program *gprof* is shown in Table 3.6. As we can see, the program uses a lot of time allocating and freeing memory. While memory management uses about 53% of the time(the top 4 functions), the `neighbor_potential()` function uses only 3.5% of the time.

Name	%Time	Seconds	Cumsecs	#Calls	msec/call
.free_y	17.2	39.09	39.09		
.global_unlock_ppc_m	17.0	38.53	77.62		
.global_lock_ppc_mp	10.4	23.70	101.32		
.malloc_y	8.5	19.20	120.52		
._lapi_shm_dispatche	6.0	13.57	134.09		
.leftmost	5.9	13.45	147.54		
.__mcount	4.2	9.53	157.07		
.grid_point_from_sub	4.2	9.48	166.55	89613450	0.0001
.splay	3.6	8.11	174.66		
.neighbor_potential	3.5	8.02	182.68	51600000	0.0002

Table 3.6: Initial output from `prof`, showing only methods consuming over 3.0% of total runtime.

Chapter 4

Optimizing PIC Codes

In this chapter, we will describe the changes made to existing codes. Section 4.1 describes the changes and extensions made to Anne C. Elster's PIC application[1]. Section 4.2 describes the modifications done to Jan Christian Meyer's PIC application[3]. The main focus is on the codes developed by Anne C. Elster.

4.1 Anne C. Elster's PIC codes

Elster's PIC code from [1] was written back in 1994 for the KSR(Kendall Square Research) shared memory system. The single-threaded version compiled cleanly after some minor changes, and a more extensive `Makefile` was created before any features were added.

4.1.1 Changes to Existing Code

The existing code implements the fourier transform to solve the field, so a natural extension was to take use of a fourier library to improve the runtime of the existing code with minimal changes. This is described in 4.1.5.

To perform tests with hybrid MPI/OpenMP we implemented a parallel SOR solver as described in Section 4.1.4.

An unsuccessful attempt to deploy the PETSc library is described in Section 4.1.6.

The virtual process-topologies used for data distribution of the matrices and communication are different for the FFTW- and SOR-solver.

Parallel Pseudo Algorithm

This algorithm describes both the FFTW and the SOR version of the simulation, changes made in this thesis involve only step 5b and 5d in the pseudo code.

1. MPI and the cartesian grid is initialized. (Different grid for FFTW and SOR version.)
2. MPI process 0 reads simulation parameters from file and broadcasts these.
3. Memory allocations for particles and field are done.
4. Initial solve step is done with all processes.
5. Main loop begins.
   (a) Field is updated locally with the charge-contributions from the particles.
   (b) Field is solved locally and globally.
   (c) Speed is computed locally.
   (d) New particle-positions are updated locally. Any particle which crosses the local field borders are migrated to a neighbor process.
   (e) If simulation is not over, go to 5.
6. Simulation finished

4.1.2 Compilation Flags and SOR Constants

When using the simulation code one must choose between the FFT solver and SOR solver at compile-time. This is done by passing the `-DPIC_FFTW` flag to choose the FFTW solver or `-DPIC_SOR` to choose the SOR solver.

The maximum error rate for the SOR solver is defined in the `declarations.h` header file, along with the border values for the SOR solver.

To enable debug output from the program, add the `-DDEBUG` flag at compile-time. This will cause the simulation to give a lot of error messages, sufficient to get a substantial slow-down.

4.1.3 Plotting of Field and Particles

By default, the position of the particles are written for each timestep to text-files in the `plotdata/` directory. If the program is compiled with the `-DDEBUG` flag, the field will also be plotted to this directory for each timestep. Bash scripts using `gnuplot` and `mencoder` have been made to visualize the output to png-images and avi-movies.

4.1.4 Making a Parallel SOR Solver

The numerical theory of our parallel SOR solver is described in 2.7. In addition to previous description, issues with parallelization and communication have to be dealt with. OpenMP is used to parallelize the solver on each process, and MPI is used for communication between processes.

Since the SOR-solver depends on updated values for each iteration, communication has to be done in each step of the iteration.

Using OpenMP Locally

Using OpenMP is simple when we have for-loops. We simply add a `#pragma omp parallel for` statement along with a list of the variables we want to be private for each OpenMP thread:

```
#pragma omp parallel for private(i,j, index)
        for(i = 1 ; i < Ny+1 ; ++i ) {
                for(j = RB_start ; j < Nx+1 ; j+=2) {
                        ... // RED-BLACK SOR update
                }
        }// end RED-BLACK SOR iteration
```

Communication and Data Distribution

The matrix is distributed in a rectangular grid as depicted in 4.1. If the global field matrix is of size (Nx x Ny), each process holds a local matrix of size (Nx/nx x Ny/ny), where nx is the number of processes horizontally and ny is the number of processes vertically in the cartesian grid of processes.

For each iteration of the solver, borders are exchanged with 4 neighbors. The processes on top and at the bottom of the grid exchanges borders with 3 neighbors. This is because the field is only periodical vertically and has fixed borders on the top and the bottom of the field.

In the RED-BLACK SOR solver, only updated values are sent, so that in a "black" iteration, the black values are communicated. When border values are received, they are stored in the ghost-area of the local matrix (marked grey in figure 4.1).

4.1.5 Using the FFTW Library

Since the FFTW library is know for its performance and ease of use, this was the first choice for a fourier solver library. The latest public available was at the time the 3.2alpha version with support for both threaded and MPI solvers for the fourier transform.

The library had to be compiled with explicit thread and MPI support by using the correct configure flags. In addition, we used the IBM thread safe compiler to compile:

```
~$ ./configure CC=xlc_r --enable-threads --enable-mpi
```

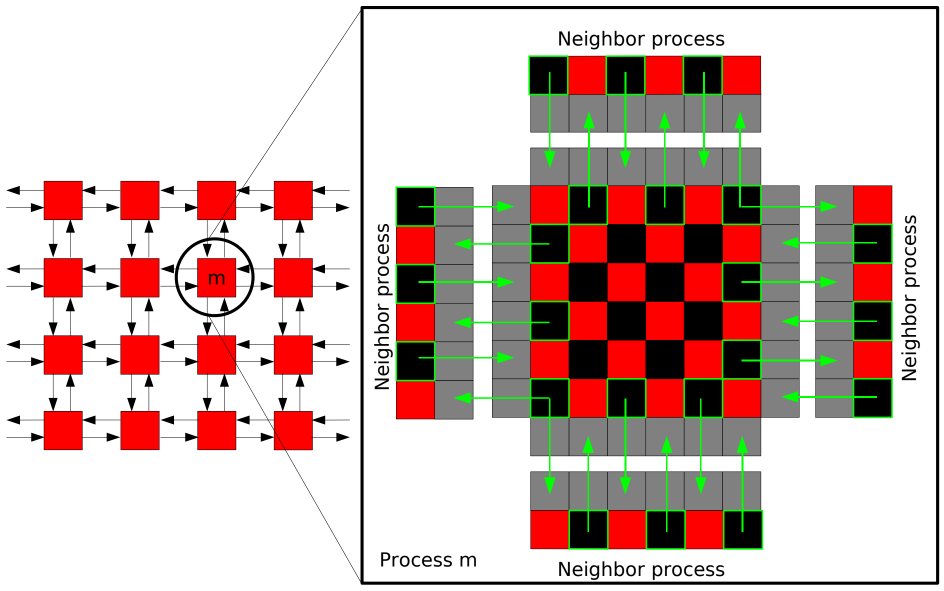

Figure 4.1: Data distribution and communication pattern in parallel RED/BLACK-SOR solver.

Using Threads Locally

The FFTW-library lives up to its reputation as an easy-to-use library, even when using the threaded version. The only code changes from the serial or MPI-version is that we add two lines of code before we set up the solver:

```
fftw_plan_with_nthreads( num_threads );
```

This configures the solver to make the solver use `num_threads` threads for each MPI process.

MPI-setup, Communication and Data Distribution

The FFTW-library is responsible for communication during the solver step, so this is hidden for the user. Communication patterns also depends which fft algorithm the FFTW-library chooses to use. The FFTW-library provides the following code to set up a MPI-parallel solver:

```
fftw_mpi_init();
alloc_local = fftw_mpi_local_size_2d(...);
data = fftw_malloc( alloc_local* sizeof(fftw_complex));
solver_forward = fftw_mpi_plan_dft_2d(..., FFTW_FORWARD );
solver_backward = fftw_mpi_plan_dft_2d(..., FFTW_BACKWARD );
```

To execute the solver, we simply state:

```
fftw_execute_dft(solver_forward, data, data);
// ... operate on the transformed data here ...
fftw_execute_dft(solver_backward, data, data);
```

The data distribution has to be the way FFTW defines if the program is to run correctly. The data distribution is very simple, and allows us to make a flat cartesian grid of size 1xN, where N is the total number of processes in the simulation. The global field matrix is divided among the processes as depicted in figure 4.2. If the global field matrix is of size (Nx x Ny), the local matrix would be (Nx/N x Ny), where N is the number of processes.

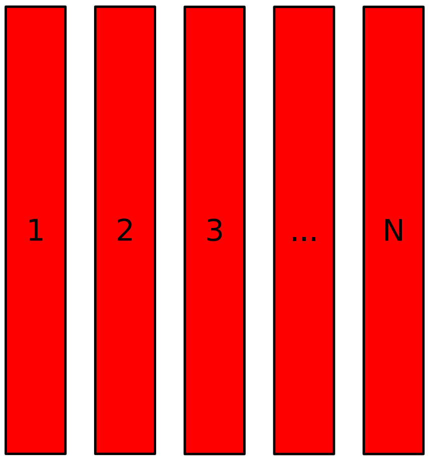

Figure 4.2: Data distribution for N processes in the MPI-parallel FFTW solver.

4.1.6 Using the PETSc Library

The PETSc library is a well known library for solving PDEs on parallel machines, but is also known to be a bit difficult to use. This was also our experience, and after about a week of trying to make it work with our existing code, we gave up. The main problem was that we could not map the local data we already had in our code into the PETSc funtcions and get the data back into our structures again after solving the PDE.

4.1.7 Migration of Particles Between MPI Processes

When a particle cross the local field border, it is sent to a neighbor process in the respective direction. This is done by the `MPI_Isend()` procedure from the sending process. All processes checks for incoming particles by using the `MPI_Probe()` function with `MPI_ANY_SOURCE` as source parameter. This is done to ensure that we receive particles from any of our neighbor processes. If the result from the probe function indicates an incoming message, the particle and its properties are received and stored in the local particle arrays. This is repeated until there are no more incoming particles. If a process has sent particles, the `MPI_Waitall()` function is used to ensure that all sent particles are transmitted successfully.

4.1.8 SOR Trace

To check if the program produces correct output, the plottings for 19 particles can be observed in Figure 4.1.8. The particles close to the zero-charged lower border, and move up to the 0.00001V charged upper border.

The particles are attracted to the positive boundary, while they are also repelled by each other, so that the trace leaves a typical SOR "fan".

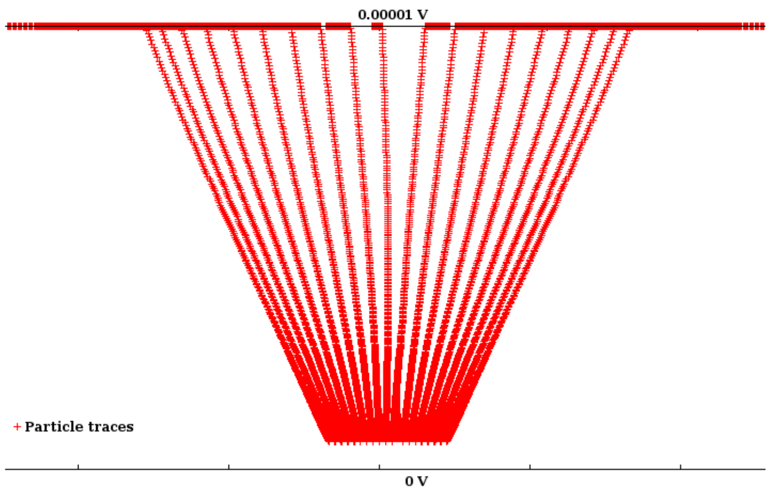

Figure 4.3: Traces of particles from a simulation using the SOR solver for 16 particles in a 0.2 x 0.2 field with 64x64 gridpoints.

4.2 Jan Christian Meyer

The code from [3] was working up to a point, but had a memory leakage as we describe in 4.2.1. As the report[3] describes, this code was left unoptimized and had a great potential of performance improvement. Performance improvements and obstacles are described in 4.2.2. Because of the complexity of the code, OpenMP or threads where not deployed in this PIC code.

4.2.1 Initial Bugfixes

In [3] Section 7.2, the code is described to sometimes saturate and crash the system. After some reading and debugging, the error was discovered to be a memory leak in one of the frequently used functions.

As memory leakages most, the bug was first found after late hours of debugging and wrapping of `free()` and `malloc()` functions. The memory leakage was due to the `subgrid_neighbor_coordinates` variable in the method `neighbor_potential()`. Memory was allocated for the variable for each time the method was called, but never freed again. This caused the program to exceed its memory limits and therefore got killed by the batch-system.

4.2.2 General Performance Improvements

Since this code was written to intentionally be a PIC simulator for N dimensions, the datatypes and solver functions had to be as general as possible regarding dimensions. This has made the code harder to read and also harder to optimize without major changes to the existing code.

Another issue is that memory is frequently allocated and freed throughout the program, and memory management takes a lot of time as shown from the initial profile in Table 3.6. By allocating and freeing memory in the innermost loops, potential performance gains might have been lost. Instead of being cpu-bound, the code is memory bound. The most important change was therefore simply to allocate most of the memory only at the start of the program, and free it again at the end of the program.

In the innermost loop in the function `neighbor_potential()`, some `if` statements and `switch-case` statements were re-written to avoid branch-mispredictions.

Chapter 5

Results and Discussion

Observations always involve theory.

Edwin Hubble

This chapter contains presentation of the performance results of the PIC simulations and discussion of these results. The emphasis is on Anne C. Elster's PIC codes.

Section 5.1 contains a brief description of the computer and processors used for the tests.

Section 5.2 contains the performance results for the FFTW solver with different ratios of MPI processes and OpenMP threads. Section 5.3 contains performance results for our parallel SOR solver with different ratios of MPI processes and OpenMP threads.

Section 5.4 explains the super-linear speedup in the results from our SOR solver. Section 5.5 describes the effect of the SMT feature in our tests. Analysis about hidden overhead in OpenMP loops and a model to estimate the communication in loops is described in Section 5.6.

In the end of the chapter we look at the results from optimizing Jan C. Meyer's PIC code in Section 5.7 and 5.8.

5.1 Hardware specifications

The specific processor we have been testing on is the Power5+ chip with 2 processor-cores with 64KB private level 1 data-cache, 1.9MB private level 2 cache and 36MB shared level 3 cache[20] as given in Table 5.1.

The system is a IBM p575 system with all in all 56 shared memory nodes of 8 dual cores and 32 GB Memory each. The interconnect between the nodes is IBM's proprietary "Federation" interconnect which provides high bandwidth and low latency. More information on the specific system can be found in IBM's whitepaper on p575 [20].

Frequency(Ghz)	1.9
L2 Latency(cycles)	12
L3 Latency(cycles)	80
Memory Latency(cycles)	220
L1 Size	64Kb
L2 Size	1.9Mb
L3 Size	36Mb

Table 5.1: Power5+ CPU specifications, quoted from [20], [21] and [22].

5.2 Performance of the FFTW solver

In this section we present the performance results from the FFTW-solver in Anne C. Elster's PIC code.

The test case is a field of size 8192 x 8192 with 32 particles. The program is run on 4 compute nodes which gives a total of 64 processors. The presented results are averages of 5 test-runs with standard deviation between 1 and 2%.

5.2.1 Efficiency

The efficiency is computed from 5 serial runs which took in average 4207 seconds. The efficiency for the FFTW solver is presented in Table 5.2 and Figure 5.1.

CPUs/Processes Threads/CPU	1	2	4	8	16
1	**0.90**	0.67	**0.49**	**0.40**	**0.29**
2	0.73	0.71	0.45	0.29	0.18
3	0.74	0.71	0.45	0.29	0.18
4	0.73	**0.72**	0.45	0.29	0.18
5	0.73	0.71	0.45	0.29	0.18

Table 5.2: Efficiency for the FFTW solver on 64 CPUs with 1-5 thread(s) per CPU and 1-16 CPUs per MPI process.

We observe that the most efficient configuration is 1 MPI process per CPU where each process uses only 1 thread. Only when we use 2 CPUs per process we get a slight speedup from increasing the number of threads.

An efficiency of 0.9 indicates that the FFTW-solver scales well on 64 processors. The difference in efficiency from the best to the worst performing configuration of the FFTW solver is 72%.

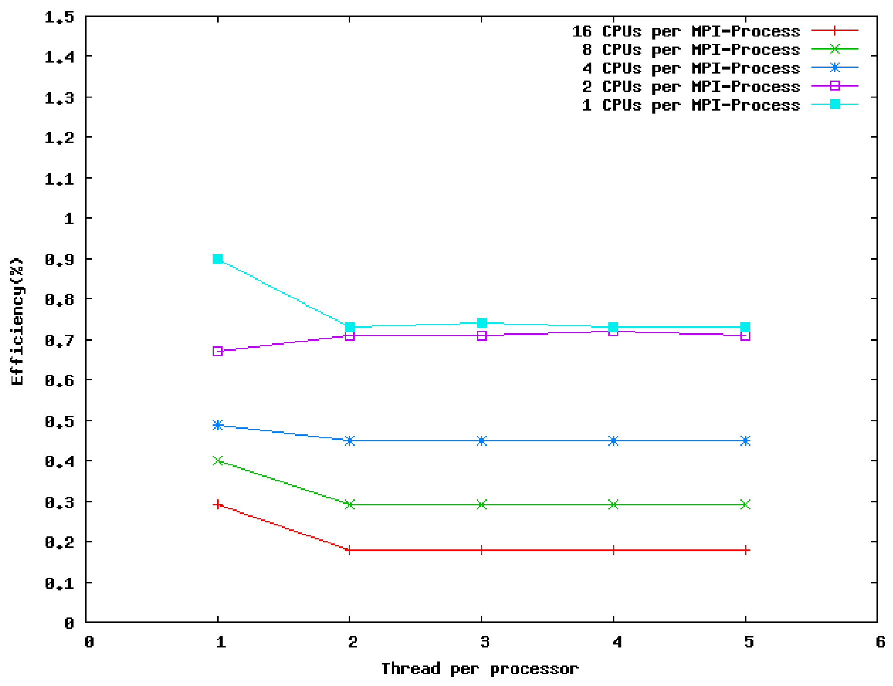

Figure 5.1: Efficiency for the FFTW solver on 64 CPUs with 1-5 thread(s) per CPU and 1-16 CPUs per MPI process.

5.3 Performance of the SOR Solver

The default test case for the program is to simulate a field of size 8192 x 8192 with 32 particles. This test case is run on 4 compute nodes which consists of total 64 processors. The presented results are averages of 5 test-runs with standard deviation between 1 and 2%.

The efficiency is computed from 5 serial run which took in average 8431 seconds.

5.3.1 Efficiency

Efficiency of the test cases with 1-6 threads per cpu and 1-16 CPU per process node is shown in Table 5.3 and Figure 5.3.

We observe that the efficiency increases for all numbers of CPUs per process when we increase the number of threads per CPU from 1 to 2. The best performing configuration is using 1 CPU per MPI process when each process use 2 threads.

With more than 2 CPUs per process, the efficiency increases with an increasing number of threads per CPU, while using 1 CPU per process will give a slowdown with more than 2 threads per process.

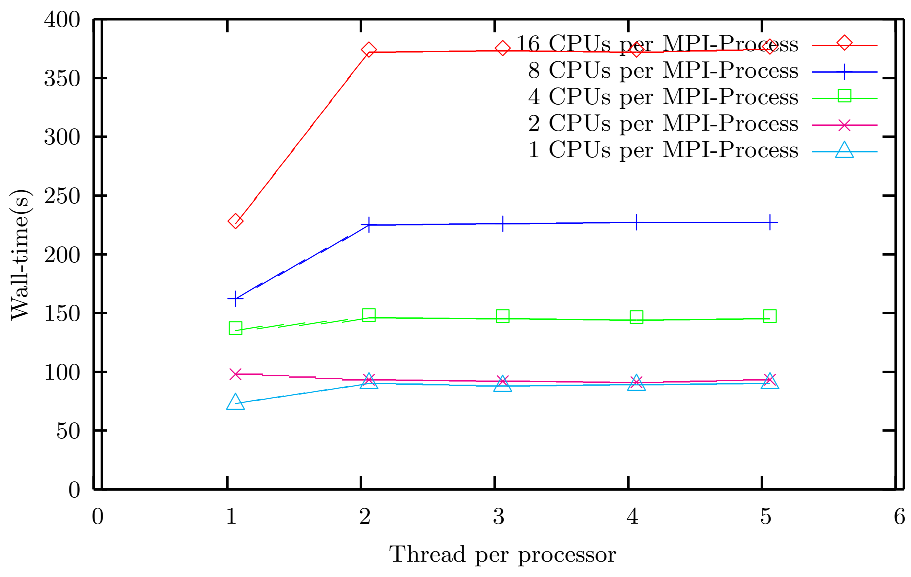

Figure 5.2: Wall-clock timings for the FFTW solver on 64 CPUs with 1-5 thread(s) per CPU and 1-16 CPUs per MPI process.

Threads/CPU \ CPU/Process	1	2	4	8	16
1	1.22	0.78	0.79	0.65	0.55
2	**1.59**	**1.49**	1.28	1.10	0.90
3	1.46	1.43	**1.49**	1.19	1.05
4	1.36	1.41	1.44	1.12	0.99
5	1.24	1.44	1.47	1.22	1.09
6	1.17	1.45	1.40	**1.24**	**1.10**

Table 5.3: Efficiency for the SOR solver on 64 CPUs with 1-6 thread(s) per CPU and 1-16 CPUs per MPI process.

From Table 5.3 and Figure 5.3 we see that our SOR solver scales well for 64 processors with the right configuration. The super-linear speedup is explained in later sections.

The difference in between the best and worst configuration for the SOR solver is 104%.

5.4 Super-linear speedup

super-linear speedup is defined as speedup greater than than the number of processor used, thus we get an efficiency greater than 1.

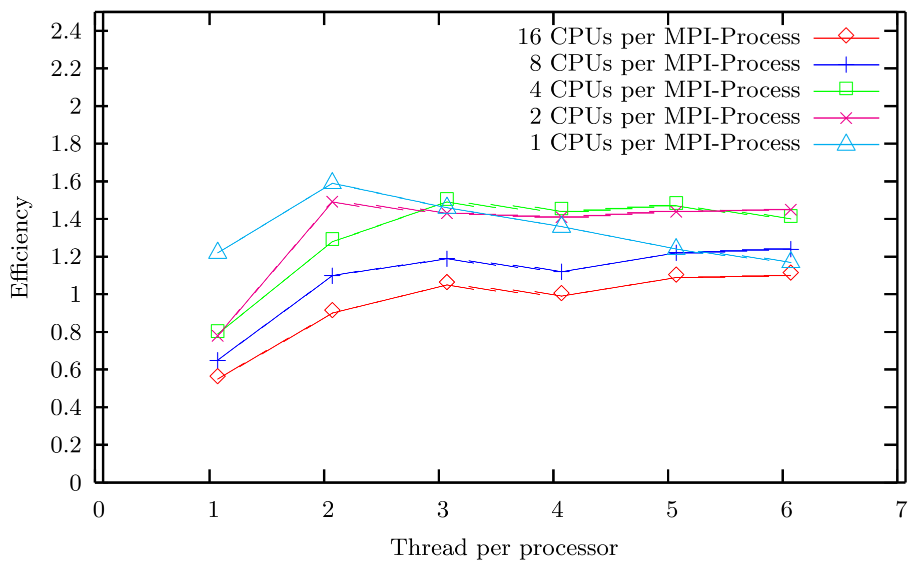

Figure 5.3: Efficiency for the SOR solver on 64 CPUs with 1-6 thread(s) per CPU and 1-16 CPUs per MPI process.

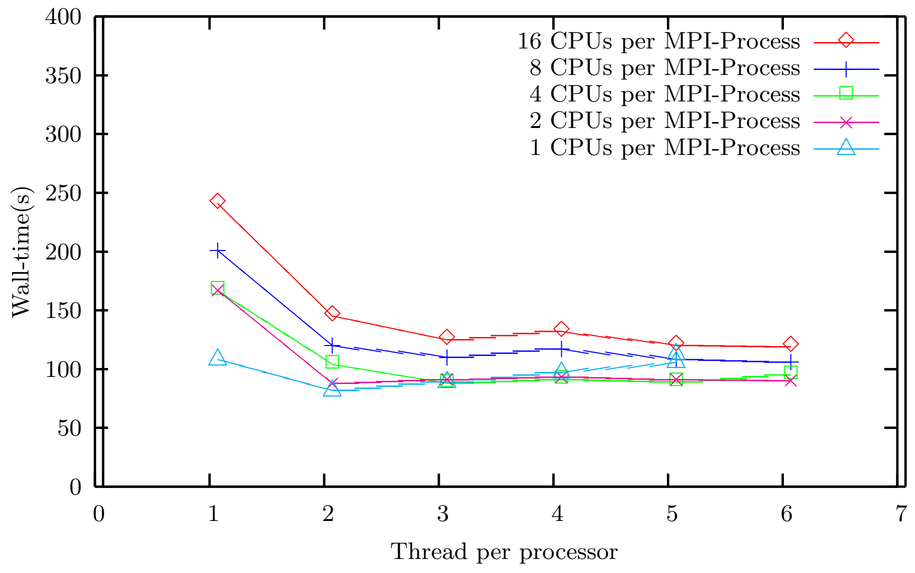

Figure 5.4: Wall-clock timings for the SOR solver on 64 CPUs with 1-6 thread(s) per CPU and 1-16 CPUs per MPI process.

There can be various reasons for super-linear speedup, one of them is the data set size compared to cache size. Cache is the fastest type of memory in a computer, and is considerably faster than regular memory. In the following subsection we consider our test case for the SOR solver and its super-linear speedup.

5.4.1 Cache hit rate

Our test case is 8192 x 8192 big. The solver function needs about

$$8192 * 8192 * sizeof(byte) Bytes = 512 MBytes \tag{5.1}$$

of space in memory. When divided on 4 nodes with 8 dual core chips, the data set on each chip is

$$\frac{512 MBytes}{4 Nodes * 8 Dualcores} = 16 MBytes/Dualcore \tag{5.2}$$

which fits in level 3 cache of the power5+ dualcore chip. From Table 5.1 we know that L3 cache has a latency of 80 cycles compared to the 220 clock cycles of the main memory. Any application that are I/O intensive would thus gain from having the entire dataset in L3 cache instead of main memory.

In addition to a speedup from the dataset fitting into level 3 cache, there are also potential gains in exploiting the size of level 1 cache for datasets with certain dimensions.

A matrix of 1048 x N double-precision floating point elements will have matrix row of 1048. A double-precision number takes 8 bytes, and therefore 3 rows of the matrix takes

$$1048 * 8 * 3 Bytes = 24576 KB \tag{5.3}$$

which fits in the level 1 cache, enabling the SOR algorithm to update at least one full row of the matrix without cache misses. When doing the SOR algorithm we use values from 3 rows(upper, lower and current row) of the matrix to update an element. In our test with global matrix size of 8192 x 8192 on 64 processors, each processor holds a 1024 x 1024 matrix, which will give super-linear speedup compared to the serial run where 3 matrix rows doesn't fit in level 1 cache. The maximum row size to fit in a 64KB level 1 cache is

$$\frac{64 * 1024}{3 * 8} = 2730 \tag{5.4}$$

. The implemented SOR algorithm will therefore theoretically favour matrices with up to about 2730 columns will therefore have an super-linear speedup on the Power5+ processor.

5.5 Simultaneous Multithreading

The Power5+ architecture provides a technology called Simultaneous Multithreading (SMT), which allows for fast thread and process switches and better utilization of the instruction pipeline.

By increasing the number of threads to a number greater than processors, several threads are waiting at the same time, and the idle time of one thread can be used by another thread to compute. This means that algorithms that causes frequent pipeline stalls or are frequently waiting for memory loads to complete, could get a speedup.

The SMT technology allows for the thread-switching to happen without the need of flushing the instruction pipeline, and we can efficiently use more of the cycles spent idling when fewer threads are used. Idling is a result of a thread not filling the pipeline with instructions, or only filling it partially.

By analyzing the inner for-loop of our SOR-solver in Listing 5.1, we derive the statistics in Table 5.4. For each update, we read 5 floating point numbers and 9 execute floating point operations. Depending on where data is located, the load operations are likely to take up several more cycles than the numerical operations. The exact filling of the pipeline and how the operations are executed is somewhat unclear since the Power5+ architecture provides out-of-order execution and each CPU can perform up to 4 floating points operations per cycle [21]. We only assume that the pipeline is not filled, and therefore we get the speedup when turning on SMT and using 2 threads per core.

Listing 5.1: Original source for the inner for-loop.

```
index = i*(Nx+2) +j;

/* load data*/
old = tempA[index];
old_left  = tempA[ index -1 ];
old_right = tempA[ index +1 ];
old_upper = tempA[ index - (Nx+2) ];
old_lower = tempA[ index + (Nx+2) ];
/* the update */
temp = ( old_left + old_right + old_upper + old_lower)/4;
temp = old + SOR_OMEGA*(temp - old);

/* error update */
err_temp = fabs(old - temp);

/* the writeback */
tempA[index] = temp;

/* err_largest */
err_largest = (err_temp> err_largest)?err_temp:err_largest;
```

Operation type	**Name**	**Count**	**Cycles/operation**
Load	`lfd` and `lfdux`	5	1/12/80/220(L1/L2/L3/RAM)
Store	`stfdux`	1	1/12/80/220(L1/L2/L3/RAM)
	Total load/store	6	
Add	`fadd`	3	6_p
Subtract	`fsub`	2	6_p
Multiply	`fmul`	1	6_p
Multiply-add	`fmad`	1	6_p
Absolute value	`fabs`	1	6_p
Compare	`fcmpu`	1	6_p
	Total numerical	9	54_p
	Total	15	

Table 5.4: Operation counts for the inner for-loop of the SOR solver. Numbers denoted $_p$ are pipelined operations.

From both Figure 5.1 and 5.3 we see that the SOR solver performs best with 2 threads per MPI process and 1 MPI process per core. For the FFTW solver, 2 threads gives a slow-down. The SOR algorithm has an O(n) analytical runtime while the FFTW has an O($nlog(n)$) analytical runtime, and therefore also has roughly $log(n)$ times numerical operations per load-operation than the SOR. We have measured this with the `pmcount` command TODO: read latest results from frigg

5.6 Communication Overhead in OpenMP

The overhead related to OpenMP is mistakenly often only related to the overhead related to spawning and destruction of threads in addition to some synchronization. The overhead from communication between the threads on the different processors is often forgotten or neglected because it is hidden from the programmer.

A node on Njord consists of 8 dual core processors and 32 GB of memory. The layout is illustrated in Figure 5.5, reproduced and simplified from [20].

When running tests with a 8192 x 8192 matrix on 4 nodes, we must use a minimum of

$$\frac{8129 * 8129 * sizeof(double)Bytes}{4nodes} = 128Mb \tag{5.5}$$

data on each node.

How the data is exactly located on each node is of not known, but assume that the matrix is located as depicted in Figure5.5, where the orange memory block contains the matrix data. Now, when our SOR algorithm uses 16 OpenMP threads, all the processors will access the memory block in all iterations of the solver. For the processors to retrieve this memory, the processors longest away from the data might have to idle several cycles to get the data to be processed. This will happen even if the data is not distributed as in Figure5.5, because of the nature OpenMP.

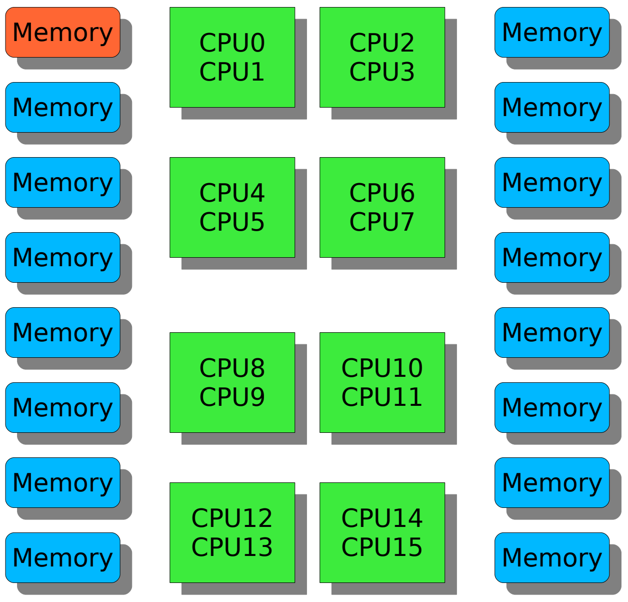

Figure 5.5: Data resides in the (orange)memory block, longest away from CPU14 and CPU15.

In the pure MPI test-runs, the data is located in a memory bank close to the processor, so the idle time when waiting for memory is avoided because of the close proximity to the memory[20]. With small data sets, the MPI-processes achieve greater performance because all of the data resides in the level 3 cache, and maximum memory latency is reduced from 220 to 80 clock cycles when accessing data.

With several threads sharing data, a significant source of overhead is introduced with communication of the data between the processors, and the overhead increases by the number of threads:

- **1 CPU per process.**
  - 1 thread. There is no overhead in communication between threads.
  - 2 threads. Shared L1, L2 and L3 cache minimizes the overhead. SMT is used so that there is no cost of context-switches of threads.
  - More than 2 threads. Overhead from thread-switching.

- **2 CPU per process**. Overhead from communication. Since the 2 CPUs are located on the same chip, communication between threads on the cores is done via the L3 cache. Threads on the same core communicates via the L1 and L2 cache.
- **4 - 16 PCU per process** In addition to the mentioned overhead, there is also overhead from communication between the dualcore chips. The communication is no longer done via cache, but via main memory or a fast databus.

The total overhead, O_t, for an OpenMP loop on processors executed by a number of threads can be expressed as

$$O_t = T_{comm} + T_{threads} \tag{5.6}$$

Where T_{comm} is the time spent in communication and $T_{threads}$ is the overhead from spawning, destroying and switching between active threads.

For a for-loop in OpenMP, a loop of i iterations is divided into chunk sizes of c, so that the number of messages sent is

$$\frac{i}{c} \tag{5.7}$$

and the size, s, of the messages are

$$s = c * sizeof(datatype) \tag{5.8}$$

.

The cost of communication of c bytes is known to be

$$T_{comm} = T_s + \beta s \tag{5.9}$$

where T_s is the latency of the respective cache level or memory where the data is located and β is the bandwidth between the processors.

Summarizing these formulas, we get a general expression for overhead of communication in an OpenMP for-loop,

$$T_{comm} = \frac{i}{c}(T_s + \beta * c * sizeof(datatype)) \tag{5.10}$$

.

There are three variables in this equation we should look into to provide a better understanding of the estimate, the bandwidth β, cache-latency T_s and chunk size c.

- β can be approximately measured with MPI-benchmarks limited to 1 and only 1 SMP node. Repeatedly broadcasts or all-to-all benchmarks with long messages can give an estimate of the intra-node-bandwidth. The bandwidth could well be related to how many processors that share communication bus, so this variable could be decreasing as the number of processors increases.

- T_s can be found with the same method as β, only with `NULL`-size messages. Depending on how many processors that participate in the computation, the length of the databus and how many processors using the databus at the same time will influence the latency. As more processors participate in the for-loop, it is likely that we end up with a decreasing average latency.

- The chunk size c can be specified statically with the OpenMP keyword `schedule(static, chunk size)`. This is however not recommended, since a normal programmer seldom knows the optimal chunk size. By not specifying the chunk size, it is set to be decided in runtime and therefore difficult to know. Finding the optimal chunk size can be done empirically by comparing different chunk sizes to the default `schedule(runtime)`.

5.6.1 Fixed Chunk Size

As mentioned in feedback[1] of a presentation of early results from this thesis [23], the overhead of communication could be reduced by setting a static chunk size to a size so that each thread in the OpenMP loop only receives data once. Then the number of messages is limited to the number of threads involved.

However, further testing with this theory indicates that we get little or no speedup in our SOR algorithm with this strategy. This could be because of border-exchanges of chunks during the calculation, or that the latency, T_s, is less significant than the bandwidth, β, in Equation 5.10.

5.6.2 Thread Affinity

In feedback[2] from [23], it was mentioned that thread-affinity could influence the performance results. If a thread migrates to an other CPU for some reason, this migration takes extra time and must be included in the overhead calculation.

To test if this was a significant source of overhead, we used the `bindprocess()` function-call to bind the OpenMP threads to a given CPU. However, this did not give any notable performance improvements. Since the `bindprocess()` function did not give any improvement, it might be that threads already are bound to a CPU in IBM's OpenMP-implementation, or that threads did not migrate before we used the `bindprocess()` call.

[1]Comments from Thorvald Natvig
[2]Comments from Jørn Amundsen

5.7 Jan C. Meyer's PIC Code

To measure the performance of the old and new PIC code, we use an increasing number of processors on a constant grid size. The results are presented in Figure 5.6 and Table 5.5. The test case used was 9 particles in a 512 x 512 field.

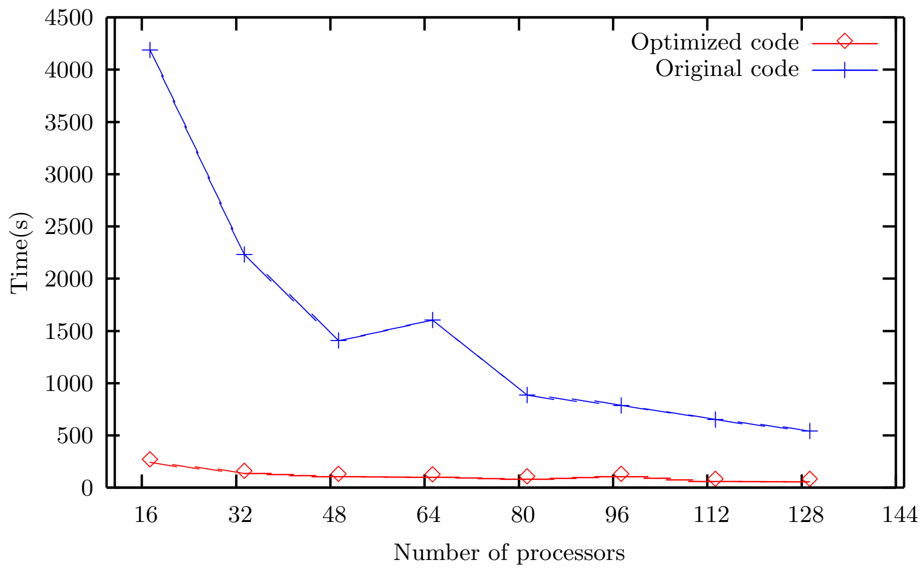

Figure 5.6: Timing results in seconds for original and optimized PIC code.

Processors	Original	Optimized	Speedup
16	4187.31	242.97	17.30
32	2231.48	135.74	16.43
48	1405.86	106.20	13.25
64	1604.26	98.32	16.31
80	887.32	81.23	10.95
96	786.59	105.37	7.48
112	653.44	59.04	11.06
128	540.14	55.84	9.81

Table 5.5: Timing results in seconds for original and optimized PIC code.

We can see from Figure 5.6 that the optimized application has only little speedup when more processors are used, even if it runs a lot faster than the original code.

Speedup from old to new code can be read out of table 5.5. We see that the new code has a substantial speedup compared to the old code.

5.8 Effects of Correct Memory Allocation in Jan C. Meyer's PIC Code

The only optimization we did to Jan C. Meyer's code was to handle memory allocations better. However, the application was from about 10 to 17 times faster after this optimization. This demonstrates only that even small optimizations can lead to significant performance improvements.

Chapter 6

Conclusion

Men love to wonder, and that is
the seed of science.

Ralph Waldo Emerson

In this thesis we parallelized a PIC simulation to study hybrid OpenMP/MPI code and evaluated the performance of different configurations of these two programming paradigms.

6.1 Contribution

In this thesis we have

1. Proposed a general model for estimating overhead in for-loops in OpenMP.

2. Did a study of OpenMP and MPI in combination to find that MPI provides the best performance of the two programming styles.

3. Parallelized an existing Particle-In-Cell simulation with OpenMP and MPI. From the results we see that the resulting application is highly scalable and efficient.

4. Optimized an existing Particle-In-Cell simulation to handle memory-allocations better.

Hybrid OpenMP and MPI Codes

To parallelize the PIC codes we have used both OpenMP and MPI to explore the possibilities of maximum performance. MPI was used globally to exchange border-values and particles while OpenMP was used to parallelize compute-intensive for-loops. While OpenMP is the easiest way to parallelize loops, our results showed that overhead from communication between the processors makes it a low-performing API compared to MPI where data is communicated explicitly.

We suggested a general expression to model the overhead of communication in OpenMP loops based on number of loop-iterations, chunksize, latency and bandwidth.

Parallelization and Optimization of PIC Codes

We converted a Particle-In-Cell code using Pthreads and optimized it with MPI and OpenMP. The results has shown that the codes are highly scalable and efficient. In addition to the original FFT algorithm, there is now options to use either the FFTW solver or our parallel SOR-solver depending on the border conditions. The parallelization includes a cartesian division of the field where all processors holds a small part of the field. Particles migrate between processors efficiently using `MPI_Isend()` when particles move from one local field to another.

We also improved the way memory was allocated and freed in Jan C. Meyer's PIC code, in addition to some other minor changes. This has lead to a 10-17 times speedup compared to the original application.

6.2 Future work

- During the project we requested for some other PIC-codes from [7], but we have still not had the chance to try this code. A natural future work is to compare the performance of such codes to our own, and see what has been done different in other versions.

- From a physics point of view, the codes could be extended to include more complex physical properties like different charged particles and more complex border boundaries.

- Further analysis on how many clock cycles are used per SOR update would be interesting to find out. However, such an analysis might require a simulation of the power5 architecture to obtain exact numbers.

- Further investigations and other programs using OpenMP should be explored to find accurate methods to estimate the overhead of using OpenMP.

- Compare the FFTW library to FFT-implementation in the IBM math library, PESSL. There might be some speedup from FFTW to the PESSL library since PESSL is optimized for the power architecture.

- The SOR solver could be optimized even more. To avoid reading and writing conflicts between the threads, we could split the field matrix into one “black” matrix and one “red” matrix. This way, all the threads needs only to read one of the matrices, and enable all threads to have a local copy of the read-matrix without any synchronization during a red/black SOR iteration.
- Load balancing is a topic we have not been looking at in this thesis. If there are many particles in a simulation, we might end up with one MPI-process with all the particles, and also all the computation related to the particles. A direct future work would be to look at how such load-balancing have been done in other simulations, and implement it in Anne C. Elster’s PIC codes.

Bibliography

[1] Anne C. Elster, “Parallelization Issues and Particle-In-Cell Codes”, 1994, Cornell University, USA.

[2] Anne C. Elster, “Software Test-bed for Large Parallel Solvers”, 1999, Iterative Methods in Scientific Computation IV, 245-260.

[3] Jan C. Meyer, “Emerging Technologies Project: Cluster Technologies. PIC codes: Eulerian data partitioning. ,” 2004, Norwegian University of Science and Technology, Norway.

[4] SCALI higher performance computing, *http://www.scali.com/*

[5] Ted Retzlaff and John G. Shaw , “Simulation of Multi-component Charged Particle Systems”, 2002, Wilson Center for Research and Techology , Xerox Corporation, USA.

[6] Thorvald Natvig, “Automatic Optimization of MPI Applications”, January 2006, Norwegian University of Science and Technology, Norway.

[7] Viktor K. Decyk and Charles D. Norton, “UCLA Parallel PIC Framework,”, January 2006, University of Los Angeles, USA.

[8] James W. Cooley and John W. Tukey, ”An algorithm for the machine calculation of complex Fourier series,” Math. Comput. 19, 297-301, 1965.

[9] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical Recipes in C, The Art of Scientific Computing*, Cambridge University Press, 1992, USA.

[10] David M. Young, *Iterative Solution of Large Linear Systems*, 461-481, 2003, Dover Publications, USA.

[11] Budi Darmawan, Charles Kamers, Hennie Pienaar and Janet Shiu, *AIX 5L Performance Tools Handbook*, August 2003, IBM International Technical Support Organization, *http://ibm.com/redbooks* .

[12] Peter S. Pacheco, *Parallel Programming with MPI*, 1997, Morgan Kaufmann Publishers, San Fransisco, USA.

[13] Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill, *Patterns for Parallel Programming*, Addison Wesley Professional , September 2004, Boston, MA, USA.

[14] Rolf Rabenseifner, “Hybrid Parallel Programming on HPC Platforms,” *EWOMP’03*, September 22-26 2003, Aachen, Germany.

[15] The Unix System – UNIX 03 - *http://www.unix.org/unix03.html*

[16] The Message Passing Interface(MPI) Standard - *http://www-unix.mcs.anl.gov/mpi/* and *http://www.mpi-forum.org*

[17] OpenMP: Simple, Portable, Scalable SMP Programming - *http://www.openmp.org/*

[18] “The Fastest Fourier Transform in The West”, FFTW - *http://fftw.org/*

[19] “Portable, Extensible Toolkit for Scientific Computation”, PETSc - *http://www-unix.mcs.anl.gov/petsc/petsc-as/*

[20] Harry M. Mathis, Frederick Bothwell, Jacob Thomas, “IBM System p5 575 8-core 2.2 GHz, 16-core 1.9 GHz Ultra-dense, Modular Cluster Nodes for High Performance Computing”, February 2006, IBM Systems and Technology Group *http://www-03.ibm.com/systems/p/hardware/whitepapers/575_hpc.html*

[21] B. Sinharoy, R. N. Kalla, J.M. Tendler, R.J. Eickmeyer J.B. Joyner, “POWER5 System Microarchitecture”, July 2005, IBM Systems and Technology Group.

[22] TOP500 Supercomputing Sites, *http://www.top500.org*

[23] Presentation of this thesis to the NTNU HPC Group, May 25th, NTNU, Norway.

[24] Anne C. Elster and David L. Presberg, “Setting Standards For Parallel Computing: The High Performance Fortran and Message Passing Interface Efforts ”, May 1993, Theory Center SMART NODE Newsletter, Vol. 5, No.3 . *http://www.idi.ntnu.no/ elster*

Appendix A

Description of Source Code

This chapter describes the functionality of each file in Anne C. Elster's parallelized PIC code.

A.1 Dependencies

To compile and run the simulation you will need the following software installed:

- POSIX compliant system. Linux and AIX 5.3 have been tested.
- `MPI` is a need for running the shared-memory version of the program.
- `FFTW` with support for MPI, this means version 3.2 or later. MPI and thread support must be compiled in explicitly with `-enable-mpi -enable-threads`.
- `OpenMP` must be supported by the compiler to use the OpenMP version of the SOR solver.
- Support for affinity should be provided by the system by some of the files in the `/usr/include/sys` directory. In AIX this is the `sys/processor.h` file.
- `gnuplot` is convenient for visualizing the output from the simulation. Most of the output is built to interact easily with gnuplot.

A.2 Compilation

Depending on the system you are on, you might want to change the Makefile. The variables `FFTW_DIR` and `MPI_HOME` should point to their respective directories. Compiler, `CC`, and compiler flags should probably also be adjusted by needs and preferences.

A.3 Source Code

A.3.1 declarations.h

- Pre-defined constants used throughout the program.
- SOR constants for Dirichlet borders, maximum error rate and maximum number of iterations the SOR solver should do.
- Global variables.
- Function declarations for all functions in the simulation.

A.3.2 mymacros.h

Contains basic function-macros made by Anne C. Elster.

A.3.3 common.c

Contains some functions that does not fit in the simulation or solver category. Debug functions and plotting functions.

A.3.4 solvers.c

Contains a wrapper function, `generic_solve()`, and all solvers. `generic_solve` chooses what solver to use from compiler flags (either `-DPIC_FFTW` or `-DPIC_SOR`). Old solvers and solvers under work can be chosen by modifying `generic_solve()`.

A.3.5 psim.c

Contains the main function of the program and all functions that calculates particles contribution and speed.

- `main` contains the main-loop of the simulation.
- `part_rho` calculates each particles contribution to the field.
- `uniform_grid_init` initializes each particles initial speed and position.
- `periodic_field_grid` calculates the field at each node in x and y direction.
- `push_v` calculates acceleration and updates speed of the particles.
- `push_loc` updates a particles location based on it speed and current location.

In addition, there are some old trace-functions in this source file we left untouched. The original trace functions of the particles uses the `hdf` format.