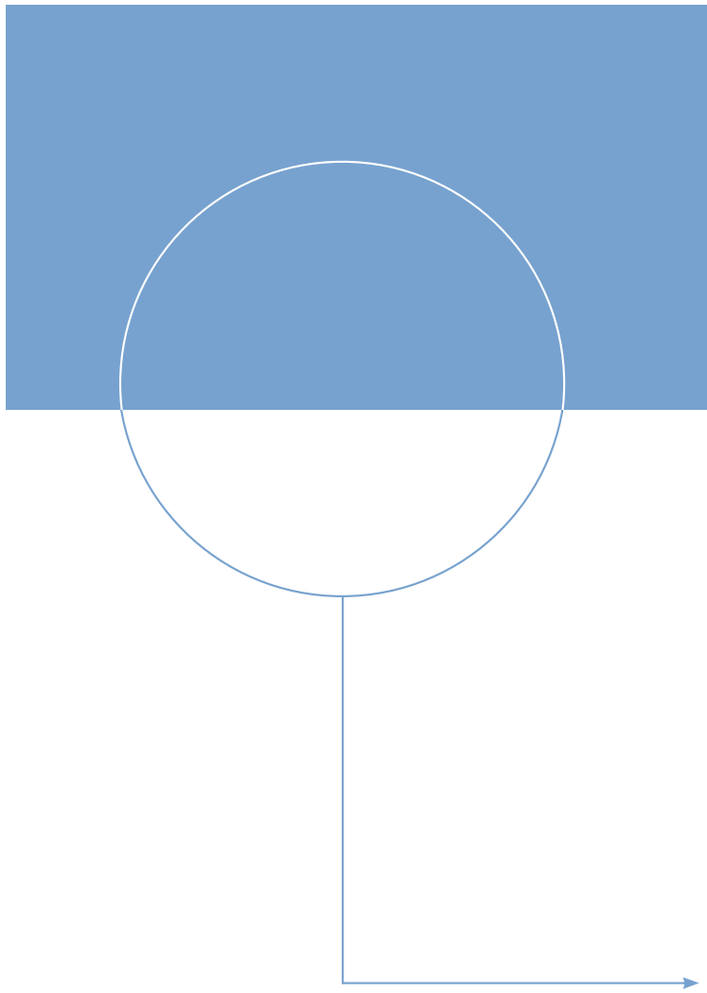


Master's thesis

2022

Nicolai Stølen

NTNU
Norwegian University of
Science and Technology
Faculty of Natural Sciences
Department of Physics

Nicolai Stølen

Distributed Memory Parallelization of a Three-Dimensional Finite Element Method Poisson Solver

December 2022

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

Distributed Memory Parallelization of a Three-Dimensional Finite Element Method Poisson Solver

Nicolai Stølen

Applied Physics and Mathematics
Submission date: December 2022
Supervisor: Jon Andreas Støvneng
Co-supervisor: Trond Brudevoll

Norwegian University of Science and Technology
Department of Physics

Abstract

For over a decade, the Norwegian Defence Research Establishment (FFI) has worked together with students from different universities to develop what is now a very capable semiconductor device simulator, FFI-MCS. It has parallelized full-band flight and scattering routines, as well as a 3D Finite Element Method (FEM) Poisson solver. As requirements for the simulator grow larger, the simulator needs to become more efficient and run faster. Parallelizing the simulator has therefore been a focus. The flight and scattering routines were parallelized by Bolstad, and now the Poisson solver is next. In this thesis an implementation of a parallelized Poisson solver was attempted.

The method of parallelization was chosen based on my project thesis, and is a distributed memory domain decomposition technique using the Message Passing Interface (MPI) to implement the parallel preconditioned conjugate gradient (PPCG) method. To do this, the mesh needs to be partitioned, and after some trial and error, it was decided that this partioning would be done through the third pary software Gmsh. A new mesh file parser had to be implemented in FFI-MCS, but after that, work on the Poisson solver could start.

Results show that the current parallel Poisson solver does not produce a valid solution. There are large errors on every partition boundary, but within each partition the solver seems to produce results that are not too deviant from normal. It is unknown as of now what causes these errors, and they will need to be researched and fixed in a future project or thesis.

Sammendrag

I over et tiår har Forsvarets Forskningsinstitutt (FFI) jobbet sammen med studenter fra forskjellige universiteter for å utvikle det som nå har blitt en veldig kapabel halvelederkomponentsimulator, kalt FFI-MCS. Den har en parallellisert rutine for partikkelbevegelse og -spredning, samt en 3D elementmetodeløser (FEM) for Poisson-ligningen. Etterhvert som kravene til simulatoren blir større, må den også bli mer effektiv og kjøre raskere. Derfor har parallellisering av simulatoren vært et fokus. Rutinene for partikkelbevegelse og -spredning ble parallellisert av Bolstad, og nå er det Poisson-løseren som står for tur. I denne oppgaven så er det forsøkt å implementere en parallell Poisson-løser.

Metoden for parallellisering av løseren er basert på prosjektoppgaven min, og er en delt minne og domeneoppsplittingsteknikk ved hjelp av Message Passing Interface (MPI) for å implementere "Parallel Preconditioned Conjugate Gradient"-metoden (PPCG). For å gjøre dette, så må nettet være partisjonert, og etter noe prøving og feiling ble det bestemt at denne delen skulle gjøres gjennom en tredjepartsprogramvare, Gmsh. En ny nettfilleser måtte implementeres i FFI-MCS, men etter det kunne arbeided med Poisson-løseren starte.

Resultatene viser at den nåværende parallelle Poisson-løseren ikke produserer riktige resultat. Det er store feil på alle overflater mellom partisjoner, men inne i hver enkelt partisjon virker det som løsningen ikke har veldig store feil. Det er ukjent per nå hva som forårsaker disse feilene, og de må undersøkes nærmere og fikses i en fremtidig prosjekt- eller masteroppgave.

Preface

This master's thesis is submitted to the Norwegian University of Science and Technology (NTNU), fulfilling the requirements of a master's thesis in Applied Physics and Mathematics at the Department of Physics (IFY). The official subject title is TFY4900 Physics, Master's Thesis.

I would like to thank my supervisor Trond Brudevoll at the Norwegian Defence Research Establishment (FFI) for his help and understanding throughout my work on the thesis. I give thanks to Professor Jon Andreas Støvneng at NTNU for assuming the role as internal supervisor. I would also like to thank all of my friends and my family for their continuous support.

Nicolai Stølen,
December 16th, 2022

Table of Contents

Abstract **i**

Sammendrag **iii**

Preface **v**

Table of Contents **vii**

List of Figures **xi**

List of Tables **xiii**

List of Listings **xiv**

Abbreviations **xix**

1 Introduction **1**

1.1 Thesis Structure 2

2 FFI-MCS **3**

2.1 Introduction 3

2.2 Previous development 4

2.3 Program Flow 4

2.4 Motivation for Parallelization of the Poisson Solver 4

3 Poisson Solver **9**

3.1 Poisson equation 9
3.2 Weak formulation 9
3.2.1 Derivation 10
3.2.2 A note on boundary conditions 11
3.3 Discretization 11
3.4 Assembly 12
3.5 Solving the system 13

4 Distributed Memory Parallelisation **15**
4.1 OpenMP 15
4.2 MPI 16
4.3 Hybrid approaches 17
4.4 Approach used in FFI-MCS 17

5 Partitioning the Mesh **19**
5.1 Different approaches 19
5.1.1 Using Gmsh's partitioner 19
5.1.2 Developing an embedded partitioner 20
5.1.3 Using the METIS library 20
5.2 Making a decision 20
5.3 Incorporating the Gmsh partitioner 21

6 Implementation **23**
6.1 Program Flow 23
6.1.1 Mesh Partitioning 23
6.1.2 Seperation of Main and Secondary Processes 23
6.1.3 Initialization Routines 24
6.1.4 Main Program Loop 24
6.1.5 Program End 25
6.2 Parallel Preconditioned Conjugate Gradient 25

6.2.1 Distributed and Accumulated Vectors and Matrices 26
6.2.2 The Algorithm 26

7 Parallel Workflow **29**
7.1 Creating the mesh 29
7.2 Partitioning the mesh 29
7.3 Running FFI-MCS 30
7.4 Post-processing 30
7.5 Summary 30

8 Results **31**
8.1 Validity 31
8.2 Runtime 36

9 Further work **37**
9.1 Fix the parallel Poisson solver 37
9.2 Optimise MPI communication patterns 37
9.3 Migrate scattering routines from OpenMP to MPI 38
9.4 Other parallelization areas 38
9.5 Improving documentation 38
9.6 Configuration and data files layout 39
9.7 Version Control 39
9.8 Persistent distributed stiffness matrix 39
9.9 Better output 39

Bibliography **41**

List of Figures

2.1 Flowchart of the main parts of MonteFFI 6

2.2 Simplified flowchart of the MonteFFI initialization procedure. 7

3.1 Example of linear basis functions on a 1D mesh with compact local support on the elements to which they belong. Each basis function ϕ_i has support only on elements i and $i+1$, where element i is the interval $[x_{i-1}, x_i]$, being exactly 0 everywhere else. 13

6.1 Separation of program flows at the start of FFI-MCS execution. . . . 24

6.2 Initialization routine for FFI-MCS. 25

6.3 Main loop flow of the main and secondary processes. To keep the chart simple enough, $t = t + dt$ has been omitted from the "No" paths for the $t = t_f$? decision blocks. 28

8.1 The electric potential at the start, in the middle and at the end of the simulation for 2 partitions. The scale at the left is the electric potential in volts. 32

8.2 The electric potential at the start, in the middle and at the end of the simulation for 4 partitions. The scale at the left is the electric potential in volts. 33

8.3 The electric potential at the start, in the middle and at the end of the simulation for 64 partitions. The scale at the left is the electric potential in volts. 34

8.4 The charge carriers of the system after the simulation has finished for different partition counts. Holes are coloured red and electrons are coloured blue. Notice how the holes seem to get trapped at the interfaces between partitions. 35

List of Tables

2.1 Summary of development history of FFI-MCS 5

List of Listings

4.1 A serial implementation of the dot product in Fortran. 16
4.2 A parallel implementation of the dot product using Fortran OpenMP. 16

List of Algorithms

3.1 The preconditioned conjugate gradient method from Fatnes' thesis [18, p. 29]. Variables are defined as in the input blocks. Furthermore, **r** is the residual vector, and the rest are working vectors and scalars. 14
6.1 The parallel preconditioned conjugate gradient method presented in Nikishkov's article [28, p. 19]. Variables are defined as in the input blocks. Furthermore **r** is the residual vector, and the rest are working vectors and scalars. Barred vectors, such as $\bar{\mathbf{z}}$, represent the accumulated form of their distributed counterpart, **z**. 27

Abbreviations

2D	Two-Dimensional
3D	Three-Dimensional
APD	Avalance Photodiode
API	Application Programming Interface
CMT	Cadmium Mercury Tellurium
CPU	Central Processing Unit
EMC	Ensemble Monte Carlo
FD	Finite Difference
FDM	Finite Difference Method
FE	Finite Element
FEM	Finite Element Method
FFI	Norwegian Defence Research Establishment (Forsvarets Forskningsintitutt)
FFI-MCS	FFI's Monte Carlo semiconductor device simulator
MCS	Monte Carlo Simulator
MPI	Message Passing Interface
PCG	Preconditioned Conjugate Gradient
PPCG	Parallel Preconditioned Conjugate Gradient
RAM	Random Access Memory

Chapter 1

Introduction

The invention of semiconductor devices may be regarded as one of the biggest turning points in human history. It has transformed how every part of our life functions and continues to do so to this day. As requirements for their performance become higher, so does the need for better and faster methods of developing them. Iterating on physical prototypes quickly becomes prohibitively expensive, and so there is a need for computer simulations to be able to predict performance and iterate on designs before any production occurs.

FFI-MCS is a semiconductor device simulator featuring a 3D Finite Element Method Poisson solver and a parallelized full band particle scattering routine. It has been in development for over 10 years, as a collaborative project between master's students and researchers at the Norwegian Defence Research Establishment (FFI). From small and humble beginnings, it has evolved into a very capable simulator.

A few years ago, FFI-MCS was a purely serial program, but as more accurate simulations are wanted, particle counts increase, grid sizes decrease, and performance take a big hit. It was therefore necessary to parallelize parts of the simulator to offset this performance hit. Bolstad parallelized the particle scattering routines in his project thesis [1], drastically reducing runtime. Now remains to parallelize the other main component of the simulator, the 3D Finite Element Method (FEM) Poisson solver.

Another motivation for further parallelizing FFI-MCS is FFI's purchase of a new machine dedicated to simulations named Freiherr. It has a 64 core CPU and 256 GB of RAM. As FFI-MCS previously had been run mainly on clusters where compute nodes had around 20 cores, this big increase in core count could yield great performance gains if proper parallelization of the code is performed.

The work done during this master's thesis was based on the conclusions reached in my project thesis [2]. There it was concluded that the 3D FEM Poisson solver should be parallelized using the distributed-memory library MPI. Any further narrowing down was not made, and so the first part of the master's thesis was finding

out what to implement.

1.1 Thesis Structure

Chapter 1 gives an introduction to the master's thesis and its motivation.

Chapter 2 explains the general workings and history of FFI's semiconductor device simulator, FFI-MCS. It gives an overview over its recent development, and its need for further development.

Chapter 3 gives a brief introduction to the Poisson equation, its weak formulation and how FFI-MCS currently solves it using the finite element method (FEM).

Chapter 4 goes over some different parallelization strategies for the Poisson solver and which approach was used in this thesis.

Chapter 5 discusses some different strategies for partitioning the mesh used by the Poisson solver, and explains the approach used in this thesis.

Chapter 6 explains how the mesh partitioning and the parallelization of the Poisson solver was implemented into FFI-MCS.

Chapter 7 gives a quick overview over the new workflow required for running FFI-MCS with a parallel solver.

Chapter 8 presents and discusses some of the results of the parallel solver.

Chapter 9 gives some recommendations on which areas of FFI-MCS that need to be improved and which ones are the most important.

Chapter 2

FFI-MCS

This chapter gives an overivew over the Norwegian Defence Research Establishment's (FFI) semiconductor device sumulator called FFI-MCS[1] before any parallelization was implemented.

2.1 Introduction

FFI-MCS is a program developed by researchers at the Norwegian Defence Research Establishment (FFI) and students from various universities. It has been continually developed for over a decade and its latest iteration, MonteFFI, supports simulation of three-dimensional semiconductor devices with a full-band parallelized particle scattering procedure and a finite element method (FEM) solver using the preconditioned conjugate gradient (PCG) method.

From small beginnings it has grown into quite the capable simulator, and has been used to simulate avalance photodiodes (APD) by researchers at FFI. The program has grown quite complex over the years, and now it needs to be optimized so that simulations don't take as much time as they do now.

In 2021, FFI purchased a machine dedicated solely to these kinds of simulations named Freiherr. It has 64 CPU cores, and so efficient parallelization of FFI-MCS is of great performance so it can take full advantage of these cores. Another such machine, Liebherr, was purchased in 2022 for the same purposes. Their names are German and mean "free man", which is kind of what these machines are meant to be - free-standing, and not reliant on any cluster or similar.

[1] "FFI Monte Carlo Simulator"

2.2 Previous development

FFI-MCS has had many different developers who have added different types of functionalities to the simulator. Their contributions have been summarised in table 2.1. Its latest iteration was made by Bolstad and is called MonteFFI [1]. It features a full-band parallelized particle scattering procedure with a 3D FEM solver.

2.3 Program Flow

The overall flow of MonteFFI is quite simple. It starts with an initialization routine, where it reads in physical parameters, time step size and other core settings. It then reads the mesh file for the physical model into a custom triangulation class, and begins assembling the stiffness matrixa and load vector. After all these initialization steps are finished, it enters the main program loop, where it solves the Poisson equation, performs flight and scattering of the particles in the system, as well as saving particle and potential data when needed.

An overview of the initialization flow of MonteFFI is presented in figure 2.2, and an overview of the main program loop flow is given in figure 2.1.

2.4 Motivation for Parallelization of the Poisson Solver

Though FFI-MCS' latest iteration, MonteFFI, is a very capable simulator, it still takes a lot of time to run a simulation. Bolstad's implementation of a parallelized flight and scattering routine remedied some of the performance issues, but there is still a lot to be desired. As the need for more complex and accurate simulations grows larger, it is more probable that smaller grid sizes are needed instead of more simulation particles. Therefore there is a strong need for a parallel Poisson solver which would cut down on execution times drastically.

Table 2.1: Summary of development history of FFI-MCS. This table is adapted from my project thesis, and based on the similar table in Bolstad's thesis [3, p. 21].

Year	Developer	Description	Ref.
2007	H. Brox	Start of FFI-MCS: A bare-bones EMC simulator.	N/A
2009	Ø. Olsen	Incorporated SCRATES for pre-calculating scattering rates, prototyped carrier-carrier scattering and Pauli exclusion modules.	[4]
	O. C. Norum	Added multiple scattering mechanisms and a proof of concept 2D FD Poisson solver.	[5]
2010	Ø. Skåring	Improved accuracy of Pauli exclusion, hot phonon and screening mechanisms.	[6]
2011	C. N. Kirkemo	Implemented an improved 2D FD Poisson solver and simulated PN-junctions and CMT APD devices.	[7]
2012	A. J. V. Vestby	Used Shockley-Ramo analysis to calculate terminal currents in single photon excited APDs.	[8]
2013	K. V. Falch	Studied Auger recombination models.	[9]
	B. Karlsen	Provided full-band tables created with $\mathbf{k} \cdot \mathbf{p}$ and ab initio methods.	[10]
	T. S. Bergslid	Enabled use of full-band structures to calculate scattering rates and selection of final states.	[11]
2014	J. Selvåg	Improved precision for selection of final states.	[12]
2015	J. J. Harang	Created a 2D FD Poisson solver for tensor grids.	[13]
2016	T. Chirac	Simulated photoconductive terahertz switches.	[14]
	D. K. Åsen	Created a self force free 2D FEM Poisson solver.	[15]
2017	D. Goldar	Studied wave function overlaps with Wien2k.	[16]
2018	M. Haug	Studied nonequilibrium Green's functions and Schrödinger-Poisson techniques.	[17]
	S. N. Fatnes	Created a 3D FEM Poisson solver.	[18]
2019	M. Estensen	Studied optimal mesh generation.	[19]
2020	A. Bolstad	Parallelized single-carrier flight and scattering.	[1]
		Incorporated Fatnes' 3D FEM Poisson solver into the full-band version of FFI-MCS, now called MonteFFI.	[3]
2022	N. Stølen	Proposed a parallelization strategy for FFI-MCS' Poisson solver	[2]

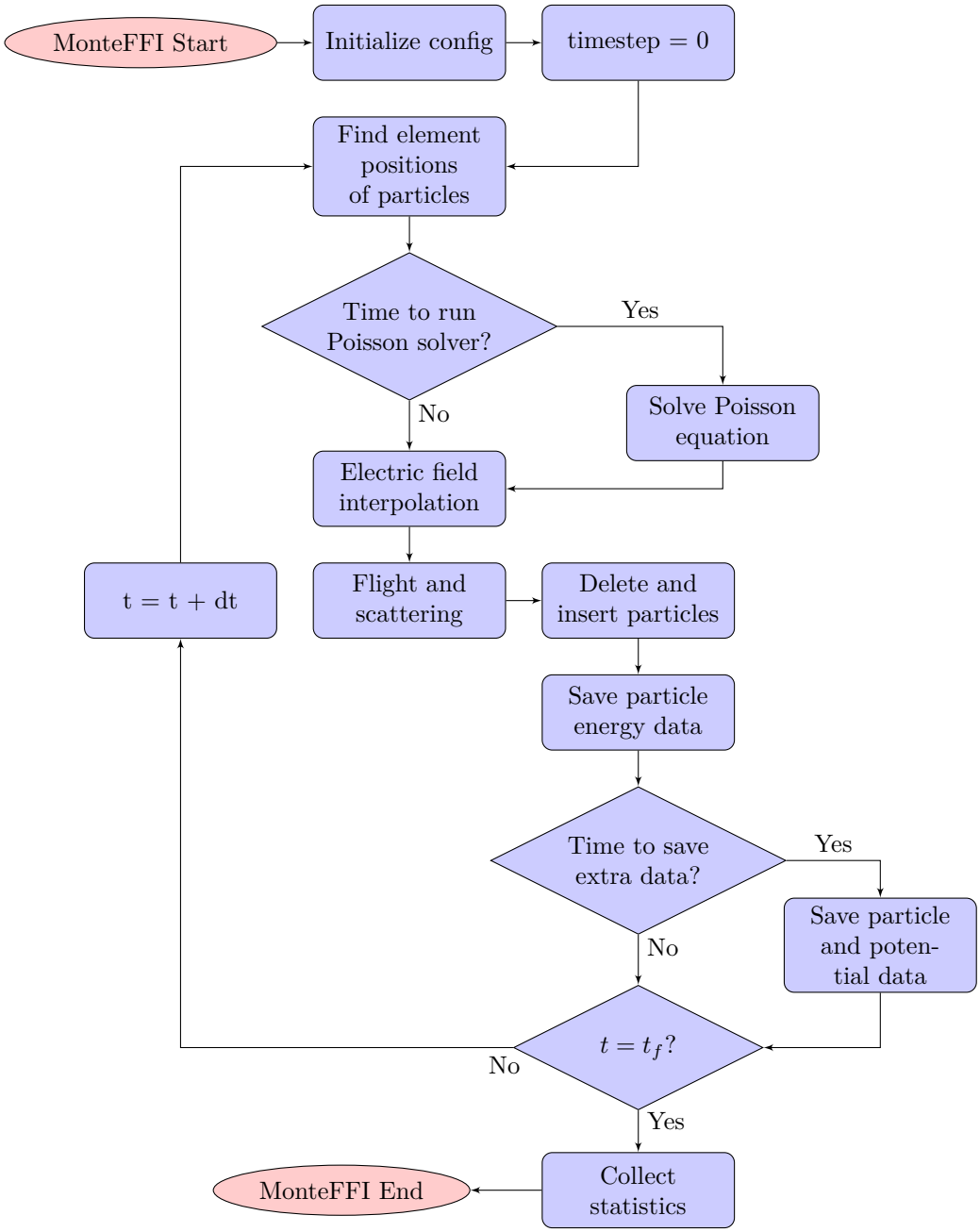


Figure 2.1: Flowchart of the main parts of MonteFFI. The Poisson solver is run every time the timestep is a multiple of the variable `poicall` which is set in the case file. Extra data about particle states and potential values are saved every 10% of the execution, as well as in the beginning. The "element position" of a particle is the element in which it is currently located. Adapted from the figure in my project thesis [2, p. 5].

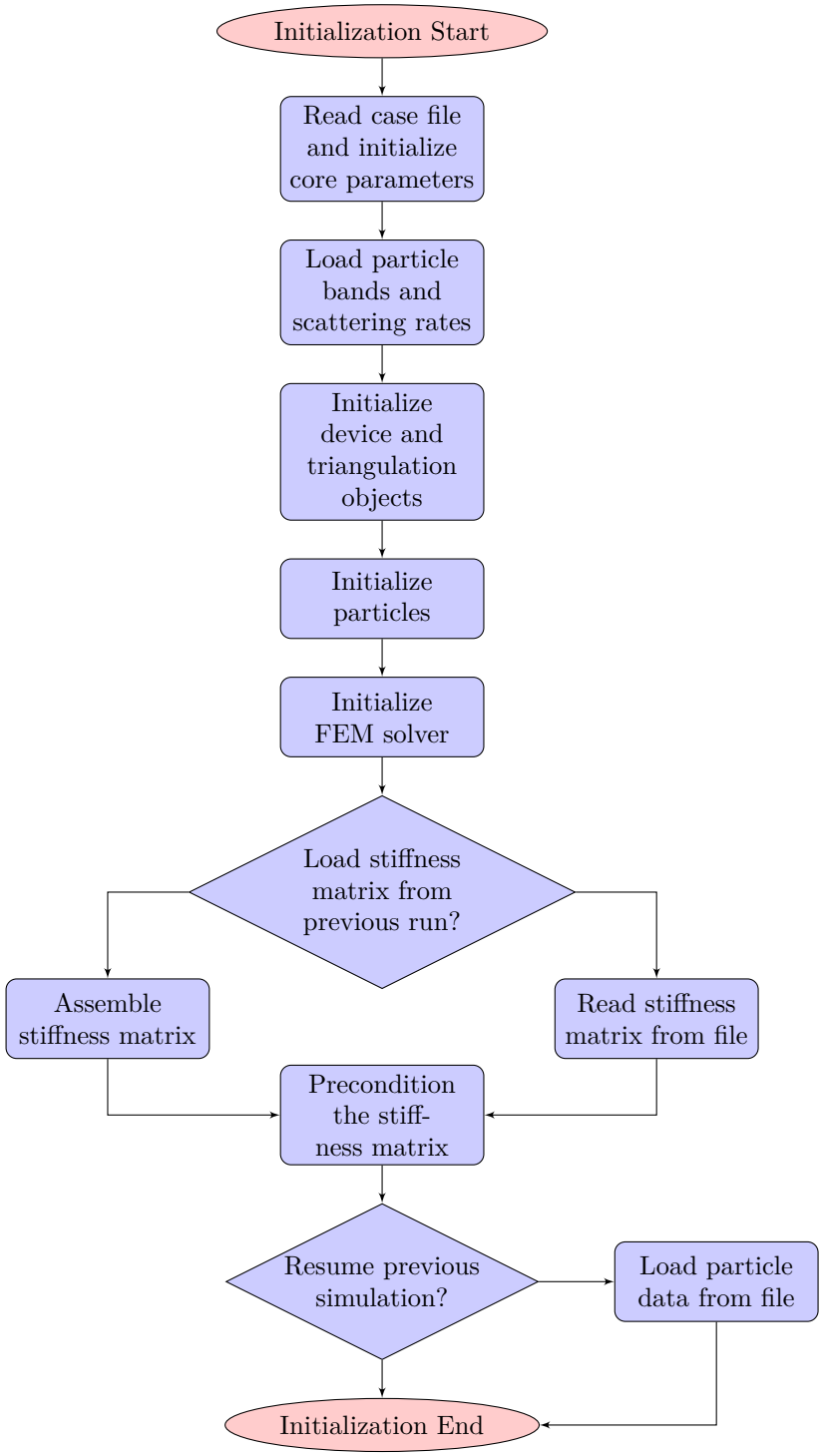


Figure 2.2: Simplified flowchart of the MonteFFI initialization procedure. Taken from my project thesis [2, p. 6].

Chapter 3

Poisson Solver

This chapter will give a quick summary of the Poisson equation, its weak formulation and the finite element method solver used to solve it in FFI-MCS. If you are not familiar with the finite element method, recommended reading material is [20, 21, 22]. For a more in-depth explanation of FFI-MCS current 3D FEM Poisson solver, see Fatnes' thesis [18, Ch. 2-3].

3.1 Poisson equation

The Poisson equation is an elliptic partial differential equation, and relates a potential function with a distribution. In our case this potential is the electric potential u, and the distribution is the distribution of charge ρ. The strong formulation of the equation

$$-\nabla^2 u(\mathbf{x}) = \frac{\rho(\mathbf{x})}{\epsilon_r \epsilon_0}, \tag{3.1}$$

is the one most people are familiar with, where ϵ_r is the relative electric permittivity and ϵ_0 is the permittivity of free space. One can discretize the differential operator in the strong formulation to solve the equation using the finite difference method (FDM), however this limits the permitted solutions u to functions (the solution space) that are twice differentiable on the domain.

3.2 Weak formulation

The weak formulation of the Poisson equation is a formulation which imposes less restrictions on the solution space. The solution in the weak formulation need only

be continuous and its derivative square-integrable.

3.2.1 Derivation

We derive the weak formulation beginning from the strong formulation with mixed homogeneous Dirichlet Neumann boundary conditions. Let Ω denote the problem domain, $\partial\Omega = \partial\Omega_D + \partial\Omega_N$ its boundary, Dirichlet boundary and Neumann boundary, respectively. For ease of notation, we set $\epsilon_r\epsilon_0 = 1$. Then our problem can be stated as

$$-\nabla^2 u(\mathbf{x}) = \rho(\mathbf{x}), \quad u = 0 \,\text{on}\, \partial\Omega_D, \quad \nabla u = 0 \,\text{on}\, \partial\Omega_N. \tag{3.2}$$

We multiply by a test function $v(\mathbf{x})$, and integrate over the domain. For simplicity of notation, function arguments will be omitted.

$$\int_\Omega -(\nabla^2 u)v \, d\Omega = \int_\Omega \rho v \, d\Omega \tag{3.3}$$

Using Green's identity, this integral is equal to

$$\int_\Omega \nabla u \cdot \nabla v \, d\Omega - \int_{\partial\Omega} v \nabla u \cdot \hat{\mathbf{n}} \, d\partial\Omega = \int_\Omega \rho v \, d\Omega. \tag{3.4}$$

The boundary integral can be split into Dirichlet and Neumann parts, giving

$$\int_{\partial\Omega} v \nabla u \cdot \hat{\mathbf{n}} \, d\partial\Omega = \int_{\partial\Omega_D} v \nabla u \cdot \hat{\mathbf{n}} \, d\partial\Omega + \int_{\partial\Omega_N} v \nabla u \cdot \hat{\mathbf{n}} \, d\partial\Omega, \tag{3.5}$$

but on the Neumann boundary, $\nabla u = 0$, so the Neumann boundary integral vanishes. For the Dirichlet boundary, we will assume that the test function v lives in the same vector space as the solution u, and therefore $v = 0$ on the Dirichlet boundary, so this integral vanishes as well. This leaves us with

$$\int_\Omega \nabla u \cdot \nabla v \, d\Omega = \int_\Omega \rho v \, d\Omega. \tag{3.6}$$

We call the term on the left hand side a bilinear form, denoted as $a(u, v)$, and the term on the right hand side a linear form, denotes $F(v)$. Using this notation, we can reformulate the problem as

$$\text{Find} \quad u \in V : a(u, v) = F(v) \quad \forall v \in V. \tag{3.7}$$

The Lax-Milgram Lemma ensures that this solution exists and that it is unique [20, p. 61].

3.2.2 A note on boundary conditions

You may have observed that this derivation was for domains with homogeneous Dirichlet boundary conditions. However, in the simulations done with FFI-MCS, we generally have inhomogeneous Dirichlet boundaries. As described in Fatnes' thesis though, we can perform a transformation between these two cases using a lifting vector [18, pp. 9-10], and so if we find a way to solve the homogeneous case, we can also solve the inhomogeneous case.

3.3 Discretization

To solve (3.7) on a computer, we need to discretize the problem. In the finite difference method, the differential operator is discretized, but in the finite element method we discretize the solution space instead. That is, instead of looking for solutions $u \in V$, we look for an approximate solution $u_h \in V_h \subset V$. The discretized problem can then be stated as

$$\text{Find} \quad u_h \in V_h : a(u_h, v_h) = F(v_h) \quad \forall v_h \in V_h. \tag{3.8}$$

This is called the Galerkin problem [20, p. 61]. Any function in this finite-dimensional subspace $V_h \subset V$ can be written as a linear combination of some basis in this space. Therefore it suffices to check that (3.8) holds for each basis function, and the problem can be rewritten as

$$\text{Find} \quad u_h \in V_h : a(u_h, \phi_i) = F(\phi_i), \quad i = 1, 2, \ldots, N_h, \tag{3.9}$$

where $N_h = \dim V_h$, the dimension of the space, and $\{\phi_i\}$ form some basis of this space. Furthermore, the solution u_h can also be written as a linear combination of the basis functions, $u_h = \sum_j u_j \phi_j$, and so the problem reads

$$\text{Find} \quad \{u_j\} : a\left(\sum_{j=1}^{N_h} u_j \phi_j, \phi_i\right) = F(\phi_i), \quad i = 1, 2, \ldots, N_h. \tag{3.10}$$

Since $a(u, v)$ is a bilinear form, this can be be written as

$$\text{Find} \quad \{u_j\} : \sum_{j=1}^{N_h} u_j a(\phi_j, \phi_i) = F(\phi_i), \quad i = 1, 2, \ldots, N_h. \tag{3.11}$$

This is a standard matrix-vector multiplication, and so we can arrive at our final formulation:

$$\text{Find}\, \mathbf{u} : A\mathbf{u} = \mathbf{F}, \tag{3.12}$$

$\mathbf{u}$ being the vector with u_j as its elements, and A being the matrix with elements $a_{ij} = a(\phi_j, \phi_i)$. We call A the stiffness matrix, and $\mathbf{F}$ the load vector.

3.4 Assembly

The stiffness matrix consists of the elements

$$A_{ij} = a(\phi_j, \phi_i) = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \, d\Omega, \tag{3.13}$$

and with the correct choice of basis functions, these elements are easily computed analytically for every i, j. A common choice of basis is linear nodal basis functions, where the basis functions are piece-wise linear functions with value 1 at one node with support only on the elements of which the node is a vertex. An illustration of such nodal basis functions in 1D is shown in figure 3.1.

As our basis functions only have support on elements containing their respective node, the integral only has contributions from these elements. We can therefore easily assemble the stiffness matrix by looping over elements and adding up the elemental contirbutions. As the basis functions are linear functions, the integral is easy to compute - their derivatives are constants.

The load vector consists of the elements

$$F_i = \int_\Omega \rho\phi_i \, \Omega, \tag{3.14}$$

and may be a little more complicated to assemble than the stiffness matrix. One has to loop over all particles in the system, distirbute their charge over nearby nodes and then sum up all local contirbutions.

The assembly procedures for the system are well-described in Fatnes' thesis [18, Ch. 2].

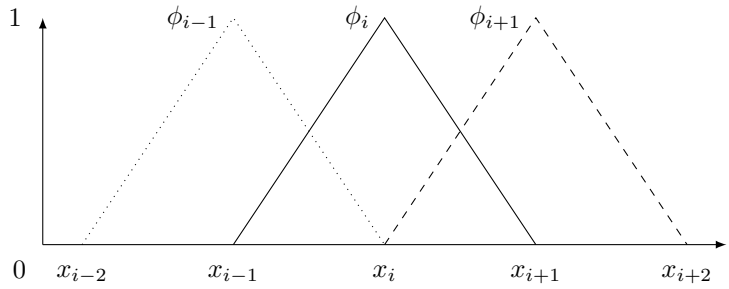


Figure 3.1: Example of linear basis functions on a 1D mesh with compact local support on the elements to which they belong. Each basis function ϕ_i has support only on elements i and $i+1$, where element i is the interval $[x_{i-1}, x_i]$, being exactly 0 everywhere else.

3.5 Solving the system

We are left with a linear system of equations and need a way to solve it in order to find the solution $\mathbf{u}$. It is possible to solve it using a direct solver, for example normal Gaussian elimination, but this often takes a very long time as the stiffness matrix can be very large, and thus this method is not feasible. The preferred approach is usually to use an iterative solver. In FFI-MCS, the solver of choice has been the preconditioned conjugate gradient method (PCG). It was implemented into FFI-MCS in Åsen's thesis [15], and is based on Saad's paper [23]. For a detailed explanation of the algorithm and its implementation, refer to the theses by Åsen and Fatnes [15, 18]. Here the algorithm from Fatnes' thesis is simply restated in algorithm 3.1.

The algorithm is one of several implemented in FFI-MCS, but Fatnes found that the PCG solver was the fastest solver [18, p. 30], and so the other solvers are no longer in use. This algorithm has a relatively easy-to-implement parallel version, which will be presented later as the main focus of this thesis.

Algorithm 3.1 The preconditioned conjugate gradient method from Fatnes' thesis [18, p. 29]. Variables are defined as in the input blocks. Furthermore, $\mathbf{r}$ is the residual vector, and the rest are working vectors and scalars.

Inputs
Stiffness matrix A
ILU0 factorization of stiffness matrix, $A^{LU} = L + U$
Load vector $\mathbf{b}$
Initial guess $\mathbf{x}_0$
Tolerance τ

Output: Solution x

$\mathbf{r}_0 = \mathbf{b} - A\mathbf{x_0}$
Solve $LU\mathbf{z}_0 = \mathbf{r}_0$
$\mathbf{p} = \mathbf{z}_0$
$i = 0$
while $||r_i||_2 \geq \tau$ **do**
    $\alpha = \frac{\mathbf{r}_i \cdot \mathbf{z}_i}{\mathbf{p} \cdot A\mathbf{p}}$
    $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha A\mathbf{p}$
    $\mathbf{r}_{i+1} = \mathbf{r}_i - \alpha A\mathbf{p}$
    Solve $LU\mathbf{z}_{i+1} = \mathbf{r}_{i+1}$
    $\beta = \frac{\mathbf{r}_{i+1} \cdot \mathbf{z}_{i+1}}{\mathbf{r}_i \cdot \mathbf{z}_i}$
    $\mathbf{p} = \mathbf{z}_{i+1} + \beta\mathbf{p}$
    $i = i + 1$
end while

Chapter 4

Distributed Memory Parallelisation

When parallelizing an equation solver, there are two main approaches taken: distributed memory and shared memory parallelization. Both have their advantages and disadvantages, and which one should choose depends upon what kind of program you are parallelizing and how efficient it needs to be. Shared memory parallelization is usually the easiest road to take, as it can often be achieved through a small amount of pre-processor statements in the code using a framework such as OpenMP. Distributed memory parallelization, on the other hand, often demands that you put more thought into communication patterns and load balancing. In some applications, the two strategies are combined to good effect.

4.1 OpenMP

OpenMP is a shared-memory application programming interface (API) for C(++) and Fortran [24]. It is usually simple to implement in a serial program, as all you need are some pre-processor statements in your code, after which the compiler takes care of the rest. Though adding these statements is easy enough, care needs to be taken to ensure the thread-safety of the routines used in these parallel sections of the code to avoid race conditions. This is the approach used by Bolstad in his project thesis to parallelize the particle scattering routines [1]. An example of a very simple implementation of the dot product in Fortran and OpenMP is shown in listing 4.1 and 4.2.

```
MODULE dotprod
    IMPLICIT NONE
CONTAINS
FUNCTION dot(a, b)
    REAL, INTENT(in) :: a(:), b(:)
    REAL :: dot
    INTEGER :: i

    dot = 0.0



    DO i=1, SIZE(a)
        dot = dot + a(i) * b(i)
    END DO


END FUNCTION
END MODULE dotprod
```

Listing 4.1: A serial implementation of the dot product in Fortran.

```
MODULE dotprod
    IMPLICIT NONE
CONTAINS
FUNCTION dot(a, b)
    REAL, INTENT(in) :: a(:), b(:)
    REAL :: dot
    INTEGER :: i

    dot = 0.0
    !$omp parallel
    !$omp do schedule(static)
↪   reduction(+:dot)
    DO i=1, SIZE(a)
        dot = dot + a(i) * b(i)
    END DO
    !$omp end do
    !$omp end parallel
END FUNCTION
END MODULE dotprod
```

Listing 4.2: A parallel implementation of the dot product using Fortran OpenMP.

4.2 MPI

The Message Passing Interface (MPI) is a standard for distributed-memory parallelization with implementations mainly for C/C++ and Fortran [25]. In an MPI program, the program is launched as several distinct processes, either on the same machine or on separate machines communicating over a network. This makes MPI very scalable, as it can be run over any number of nodes, as opposed to OpenMP which is limited by the number of processor cores of the machine it is running on. With careful design of the data structures used and the communication patterns between the MPI processes, very efficient parallelized programs can be made, typically with better performance compared to a shared-memory parallelization. This performance gain over shared-memory implementations comes from e.g. better cache locality, and faster memory access as the processors do not share memory.

An implementation of the dot product in MPI is much more involved than in OpenMP, and the code would be much longer, and an example of this is therefore omitted from this section.

4.3 Hybrid approaches

In some case shared-memory and distributed-memory parallelization using MPI and OpenMP are combined. For example you may distribute some calculations which are loosely coupled between some sub-domains over several computers using MPI and then performing some more tightly coupled computatations within the subdomains on each machine using OpenMP. An implementation using this approach is presented by Guo et al. in [26].

4.4 Approach used in FFI-MCS

Up until now, the only parallelization approach used in FFI-MCS is shared-memory parallelization using OpenMP. It is used in the parallel flight and scattering routines developed by Bolstad in his project thesis [1]. To develop the parallel Poisson solver, MPI was the tool of choice, as a domain decomposition technique was to be used, which with an efficient partitioning of the mesh reduces the need for communication between processes, and so there is more room for optimization using MPI. In the future, one might look at migrating the flight and scattering routines from OpenMP to MPI as this might yield performance benefits as well.

Chapter 5

Partitioning the Mesh

In order to effectively use a domain decomposition method to parallelize the solver, we need to partition the problem mesh into several sub-meshes. This chapter will go through the options that were considered, and the approach that was ultimately chosen.

5.1 Different approaches

Some different approaches to partitioning were considered, in particular using the built-in partitioner in the already-used mesh software Gmsh, developing a embedded into FFI-MCS and incorporating the third-pary library METIS into the code. A summary of the different approaches and their benefits and drawbacks follows.

5.1.1 Using Gmsh's partitioner

Gmsh is the meshing software already used to create models for simulations in FFI-MCS. Included in the Gmsh distribution, is a few different partitioners, for example the widely-used library METIS [27]. Anyone using FFI-MCS should already have Gmsh installed on their machine, and so this would be a very natural approach. Implementing the partitioned mesh into FFI-MCS would not be very difficult either, as the mesh file format does not differ greatly between partitioned and unpartitioned meshes.

However, there have been signals from the researchers at FFI about not wanting to interconnect FFI-MCS and Gmsh too much, as this would make the code even more reliant on Gmsh - which is, when all comes to all, a third-party dependency. They are also looking at the possibility of shipping an in-house mesher to use with FFI-MCS, and so changes to the file format it uses should be kept to a minimum.

A benefit of "outsourcing" the partitioner to a third party is that developers of

FFI-MCS would not need to maintain even more code. This extra feature would also render the whole program more complicated, and if it is tightly couples to the rest of the program code, the maintainability of FFI-MCS as a whole could suffer.

5.1.2 Developing an embedded partitioner

Another possibility is developing a meshing procedure which will automatically partition any mesh supplied in a simulation to the correct amount of sub-meshes based on the number of parallel processors. This would abstract the partionining end of things away, so that the end user would not need to think about it. This was the preferred approach of the researchers at FFI.

5.1.3 Using the METIS library

Instead of using METIS through Gmsh, one could the library into FFI-MCS. This, however, arguably couple FFI-MCS even more tightly to third-party dependencies. Not only would it be dependent on a third-party software, but a specific version of one library. Changes in either Gmsh or METIS could then render the simulator broken if great care is not given to which versions are used. As METIS is implemented in C, one would also need to use bindings between Fortran and C code, further complicating the code.

5.2 Making a decision

After some explorative work, it was decided to either go for Gmsh' partitioner, or to implement an embedded partitioner. Using METIS in the FFI-MCS code did not seem like a viable option because of the complexity of involving C bindings in the code and the maintainability hit FFI-MCS would take.

The signals from FFI was that the embedded partitioner was the preferred approach. Work was therefore started to implement this it, at first exploring different partitioning algorithms. Among the algorithms considered were recursive graph bisection and recursive graph labeling. I decided to go for the graph labeling algorithm presented by Nikishkov [28], as I was going to use his parallel preconditioned conjugate gradient algorithm. A great amount of work was put into implementing the the partitioner, but after some time it was apparent that it was taking too much time and would negatively affect the chances of implementing the parallel solver in time. Therefore a choice was made to abandon this effort and instead use Gmsh' partitioner.

5.3 Incorporating the Gmsh partitioner

Incorporating the Gmsh partitioner involves modifying both the procedures for reading in the mesh file to support partitioned meshes, and implementing support for this partitioned mesh in the triangulation routines and the Poisson solver. When first researching the support for partitioned meshes in Gmsh, it seemed as if support for a newer mesh format, MSHv4, was needed in place of the older version, MSHv2, which was in use by FFI-MCS. Therefore this was the version implemented into FFI-MCS, resulting in a major rewrite of the mesh reading code. After this work had more or less concluded, it was discovered that this was incorrect, and that the older MSHv2 format also supported partitions.

The newer MSHv4 format does however have some benefits over the old format which may be beneficial to a parallel solver based on domain decomposition. When a partitioning was done in the MSHv2 format, the partition number of each element was simply appended to every element definition line. In the newer format, elements and nodes are grouped into entities. These entities may be points, curves, surfaces or volumes. For a non-overlapping domain decomposition like we are using, nodes in a volume entity belong to only one partition, while nodes in a point, curve or surface entity may belong to several partitions - these are the boundaries between partitions.

Because of this grouping, designing efficient communication patterns between the processes may be much easier. You can group all the information about nodes lying on a partition boundary together and send it with a single MPI call easily. This is in contrast to the old mesh format, where you would have to implement some custom method of detecting interfaces between partitions and assembling interface node values to send.

Chapter 6

Implementation

Implementing a parallel Poisson solver for FFI-MCS required a lot of work. First of all, a choice had to be made about how to parallelize the solver. Some parallelization schemes require more work than others to implement, while other solutions are quite simple to implement, but might not yield the same performance benefits. A lot of time was also spent trying to implement an embedded partitioner before that approach was abandoned, which limited the time available to implement a proper solver.

6.1 Program Flow

This section will give an overview of the new program flow for FFI-MCS with the parallel Poisson solver.

6.1.1 Mesh Partitioning

Mesh partitining is not a part of the main program flow, but done externally using Gmsh. However, an implementation of an embedded mesh partitioner was attempted early in the thesis. It was abandoned after concluding that it would take too much time.

6.1.2 Seperation of Main and Secondary Processes

As the secondary MPI processes in FFI-MCS at the moment do nothing but partake in solving the Poisson equation, their program flow is quite divergent from the main process. Therefore, their whole lifecycle has been put in its own module. At the start of the program execution, all secondary processes are sent to this module instead, where they await instructions from the main process. This is a very simple

check, just checking whether the process' MPI rank is 0 or not, as shown in figure 6.1.

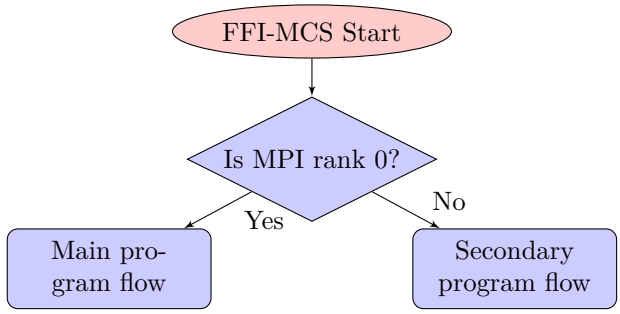


Figure 6.1: Separation of program flows at the start of FFI-MCS execution.

6.1.3 Initialization Routines

For loading physical parameters, the mesh itself and other settings for the simulation, one could opt for only the main process reading and initializing the configuration, all processes doing so, or a mix of the two. In FFI-MCS, the first option is the one used. The main process reads in all of the simulation settings, as well as the partitioned mesh itself, and afterwards broadcasts necessary data to all other processes using MPI routines.

FFI-MCS has a mix of primitive data arrays, which are easily sent using MPI, and some composite data structures where one either has to make a new MPI type and broadcast that, or make a routine for the specific structure for broadcasting its data. The latter is what was opted for in FFI-MCS. Structures such as one holding essential device parameters as well as the structure holding the triangulation data have subroutines which use blocking MPI broadcast routines to send data.

A visualization of the initialization flow is shown in figure 6.2.

6.1.4 Main Program Loop

After initialization, FFI-MCS enters a loop which lasts until the given number of time steps have been executed. In the main process, this includes solving the Poisson equation, interpolation of the electric field, calculating particle scattering, injecting and removing particles from the system and saving statistics. The secondary processes' loop consists only of solving the Poisson equation however. Therefore they will solve the Poisson equation, and then wait until the main process reaches the Poisson solver code section again. An chart of this flow is shown in figure 6.3.

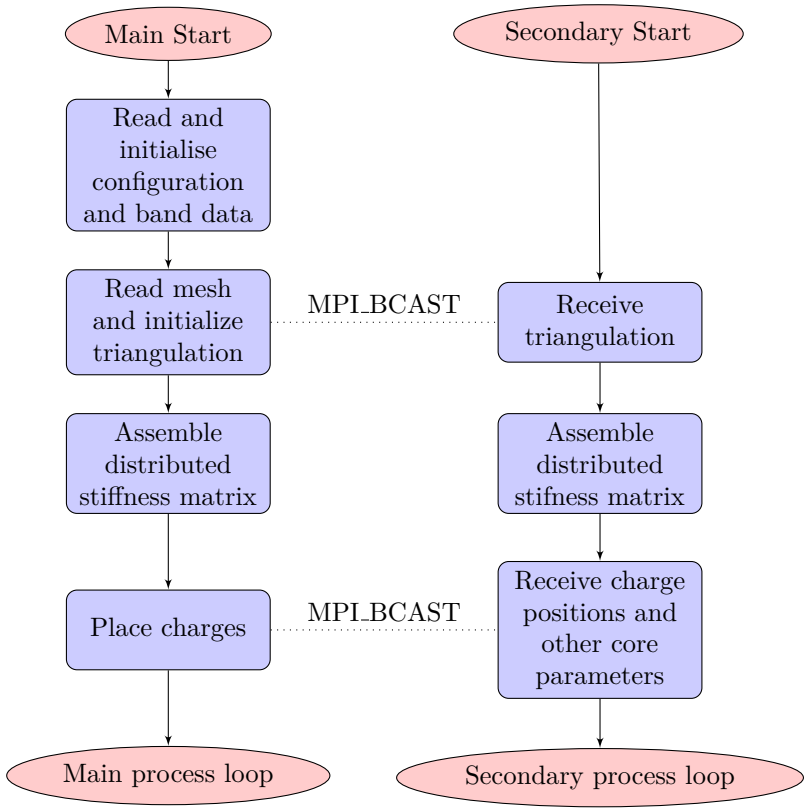


Figure 6.2: Initialization routine for FFI-MCS.

6.1.5 Program End

When the simulation has concluded, the secondary processes simply exit, while the main process saves various data and information, such as execution times. This is unchanged from previous versions of FFI-MCS.

6.2 Parallel Preconditioned Conjugate Gradient

The algorithm of choice to parallelize FFI-MCS was the parallel preconditioned conjugate gradient method (PPCG). This was because the serial version, the preconditioned conjugate gradient method (PCG) was already in use in FFI-MCS. It therefore seemed natural to go for its parallel version, as it seemed relatively easy to implement.

The preconditioned conjugate gradient method uses only some matrix-vector and vector-vector products when computing the solution. As these products are easily

parallelizable, and since the preconditioned conjugate gradient method is already the primary solver in use in FFI-MCS, it seemed like a promising parallelization route. The parallelization algorithm has been adopted from Nikishkov's article [28], and restated here in algorithm 6.1.

6.2.1 Distributed and Accumulated Vectors and Matrices

In his algorithm, there is a distinction between distributed and accumulated vectors and matrices. An accumulated vector or matrix is one which contains contributions from all partitions. A distributed vector or matrix is one which contains only contributions from the current process' partition. That is, if a node is part of elements of different partitions, its distributed stiffness matrix entry contains only contributions from the current process' partitions. The accumulated versions of these matrices and vectors will then simply be the sum of the distributed versions.

To calculate the distributed stiffness matrix and load vectors for every process, simply skip all elements which do not belong to the process' partition during assembly. If the accumulated form is needed, this can be assembled during computation, using MPI routines.

6.2.2 The Algorithm

The algorithm is taken from Nikishkov's paper [28], and is presented in algorithm 6.1. Comparing this algorithm to the serial version in algorithm 3.1, they are quite similar, bar the communication calls in the parallel version.

The preconditioned conjugate gradient method uses only matrix-vector products and inner products of vectors. Nikishkov shows in his article that for matrix-vector products, you need a distributed matrix and an accumulated vector, and for inner products, you need one accumulated vector and one distributed vector [28, pp. 16-17].

Algorithm 6.1 The parallel preconditioned conjugate gradient method presented in Nikishkov's article [28, p. 19]. Variables are defined as in the input blocks. Furthermore $\mathbf{r}$ is the residual vector, and the rest are working vectors and scalars. Barred vectors, such as $\bar{\mathbf{z}}$, represent the accumulated form of their distributed counterpart, $\mathbf{z}$.

Inputs
Stiffness matrix A
Preconditioner M
Load vector b
Initial guess x_0

Output: Solution x
$r_0 = \mathbf{b} - A\mathbf{x_0}$
Send $\mathbf{r}_i^b$, receive $\mathbf{r}_i^{eb}$, $\bar{\mathbf{r}}_i = \mathbf{r}_i + \mathbf{r}_i^{eb}$
for $i = 1, 2, \ldots$ **do**
    Solve $M\mathbf{z}_i = \bar{\mathbf{r}}_i$
    $\gamma_i = \bar{\mathbf{r}}_i^T \mathbf{z}$
    Reduce γ_i
    Send $\mathbf{z}_i^b$, receive $\mathbf{z}_i^{eb}$, $\bar{\mathbf{z}}_i = \mathbf{z}_i + \mathbf{z}_i^{eb}$
    if $i = 1$ **then**
        $\bar{\mathbf{p}}_i = \bar{\mathbf{z}}_i$
    else
        $\bar{\mathbf{p}}_i = \bar{\mathbf{z}}_i + (\gamma_i/\gamma_{i-1})\bar{\mathbf{p}}_{i-1}$
    end if
    $\mathbf{z}_i = A\bar{\mathbf{p}}_i$
    $\beta = \bar{\mathbf{p}}_i^T \mathbf{z}_i$
    Reduce β
    Send $\mathbf{z}_i^b$, receive $\mathbf{z}_i^{eb}$, $\bar{\mathbf{z}}_i = \mathbf{r}_i + \mathbf{r}_i^{eb}$
    $\bar{\mathbf{u}}_i = \bar{\mathbf{u}}_{i-1} + (\gamma_i/\beta_i)\bar{\mathbf{p}}_i$
    $\bar{\mathbf{r}}_i = \bar{\mathbf{r}}_{i-1} - (\gamma_i/\beta_i)\bar{\mathbf{z}}_i$
    if $\gamma_i/\gamma_0 \varepsilon$ **then**
        Convergence criteria satisfied: exit
    end if
end for

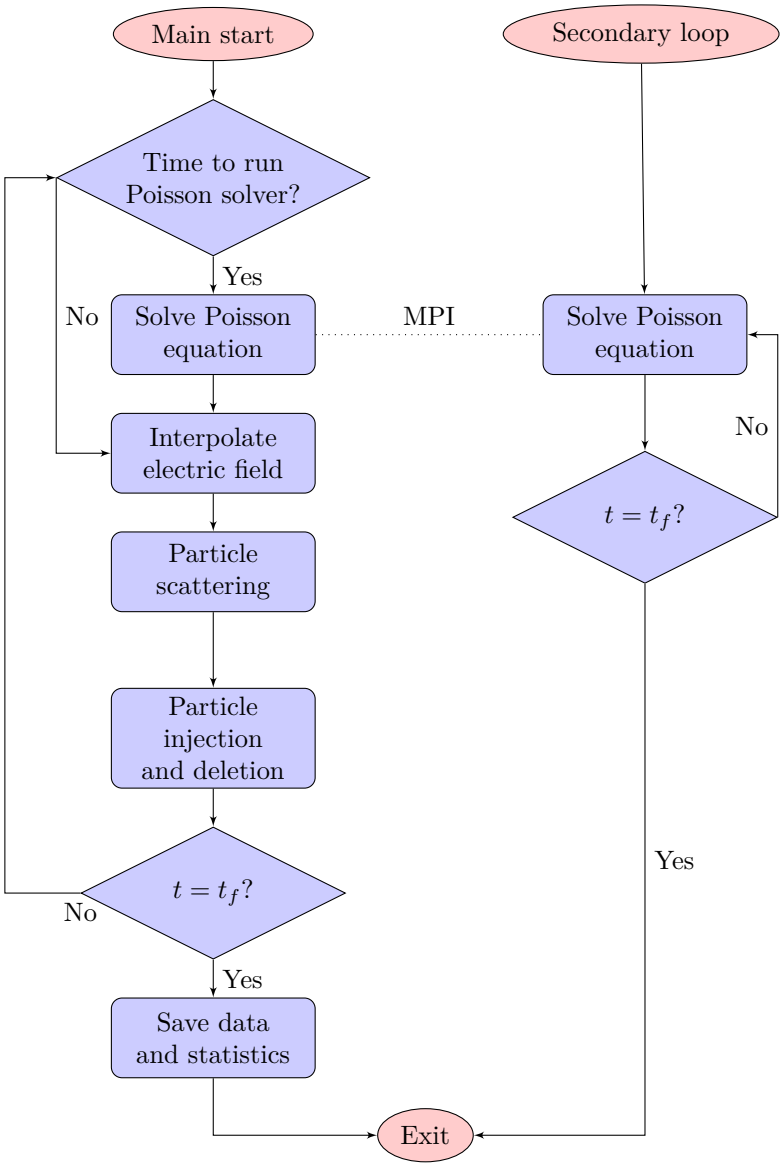


Figure 6.3: Main loop flow of the main and secondary processes. To keep the chart simple enough, $t = t + dt$ has been omitted from the "No" paths for the $t = t_f$? decision blocks.

Chapter 7

Parallel Workflow

Migrating from the serial version of the Poisson solver in FFI-MCS to a parallel one also involves changing the workflow when using the simulator. This chapter gives a quick overview over the workflow one might use for a parallel solver.

7.1 Creating the mesh

This step will still be the same as in earlier versions of FFI-MCS. However, one must take care to define the mesh file format as version 4.1, and not 2.2 as before. This is because the mesh file reading routines in FFI-MCS no longer support this old mesh file format, as the newer one supports partitioned meshes in a better way than the older versions.

For a guide on how to create a mesh for FFI-MCS, read the user guide made by Bolstad in his master thesis [1, pp. 81-83].

7.2 Partitioning the mesh

The second step will be to partition the mesh inside of Gmsh. This is when the user needs to decide on the number of processors they want to use. Usually this will be equal to the core count of the machine, or the combined core count of all nodes used in a cluster. However, there may be diminishing returns for increased core counts, and if there are several simulations running on the same machine, the user may want to limit the core count to share resources across simulations.

The choice of partition count will also depend on how fine the mesh is. Using a lot of partitions for a relatively coarse mesh will not necessarily result in the wanted performance gain as communication overhead between processes will become large.

Partitioning the mesh can also be done from the command line, as for example `gmsh -3 -order 1 -o mesh.msh -part PARTITION_COUNT geometry.geo`, where `PARTITION_COUNT` is the number of partitions wanted, and `geometry.geo` is your Gmsh model file.[1]

7.3 Running FFI-MCS

FFI-MCS will need to be run using MPI. The user should launch a simulation using MPI and a processor count equal to the partition count. To launch a simulation using 20 cores for flight and scattering routines, and 32 cores for the Poisson solver, you can use

```
OMP_NUM_THREADS=20 mpirun -np 32 bin/MonteFFI cases/model/settings.cfg
```

On some distributions, such as on Freiherr[2], you might have to use `mpirun.openmpi` instead of `mpirun` if FFI-MCS is compiled with `gfortran`.

7.4 Post-processing

This step will be identical to before. There is no difference in the format of the outputted particle positions, energies and potential values. If you use the mesh itself to plot values, you have to make sure your tool of choice supports the newer MSHv4 format, and not just MSHv2.

7.5 Summary

The workflow will in many ways be similar to before, but the user needs to take care in evaluating how many partitions to make against how many processor cores they have at their disposal and how fine or coarse the mesh is.

[1] `-3` means 3 dimensions and `-order 1` means first order basis functions, that is, linear ones.
[2] FFI's dedicated simulation machine

Chapter 8

Results

The implementation of the parallel Poisson solver was unfortunately not finished in this thesis, and so performing any performance and efficiency benchmarks would not be of great use. Therefore this section will primarily discuss some of the erroneous results and what may be possible causes.

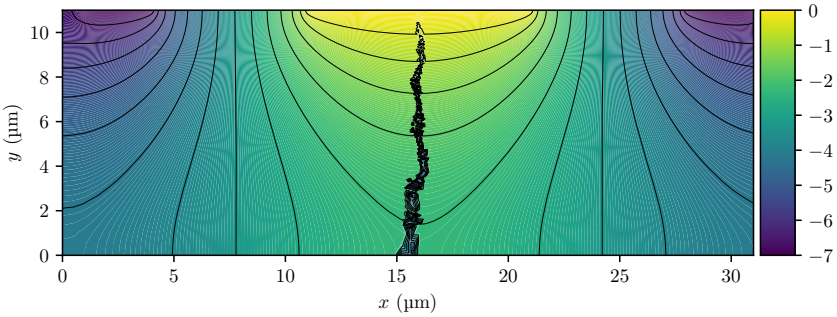
It may seem like the solver actually succeeds in calculating the potential locally, but the combination between subdomains fails, giving rise to the problems of particles falling into the "pit" and the potential looking very incorrect.

8.1 Validity

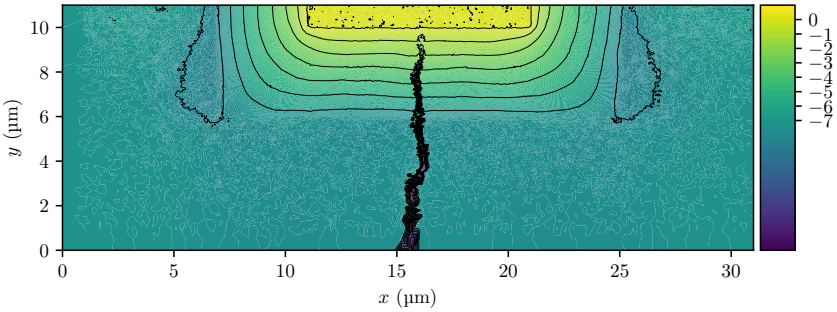
Perhaps the most important thing for a solver is to produce accurate results. The parallel solver currently does not seem to work correctly, and the results it spits out are not accurate. By looking at figures 8.1, 8.2 and 8.3, we can see that the solver seems to fail at the boundaries between partitions. This is perhaps the most common type of error to manifest itself in such a parallelized solver, and is per now not known what causes it. After trying to debug the solver for a long time, even trying the non-preconditioned PCG method to see if there was a problem with preconditioning, it still produces erronous solutions.

One thing which is interesting though, is that even though it seems to fail quite drastically at the interfaces between partitions, the solution inside each partition on its own seems not too far off. However, this is just by looking at the plots, and more digging needs to be done to find out what is producing these erronous results.

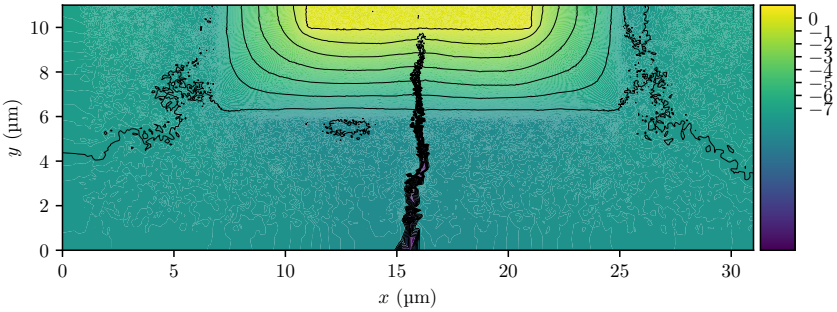
These incorrect solutions to the electric potential of course also affects the flight and scattering routines. The holes in the system are drawn into the "boundary pits" and get stuck there. This is illustrated in figure 8.4.



(a) Start of simulation

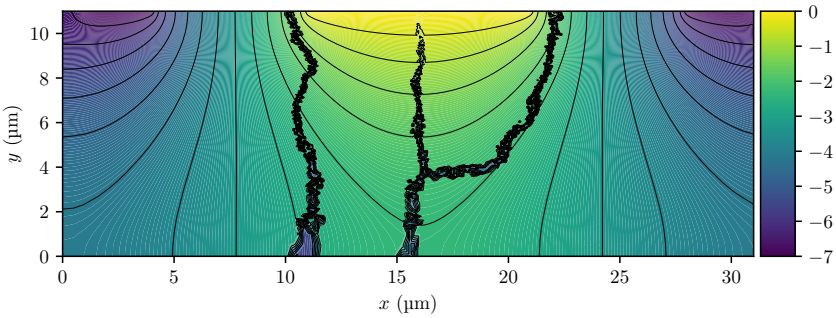


(b) After 50% of runtime

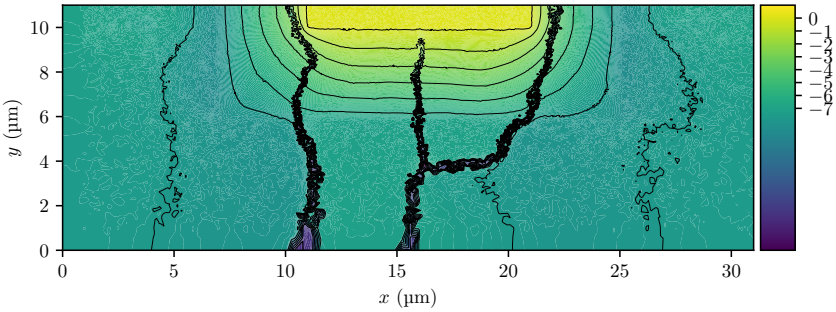


(c) At end of simulation

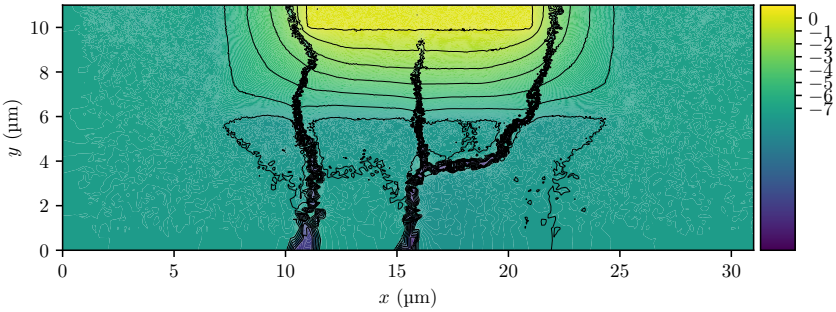
Figure 8.1: The electric potential at the start, in the middle and at the end of the simulation for 2 partitions. The scale at the left is the electric potential in volts.



(a) Start of simulation

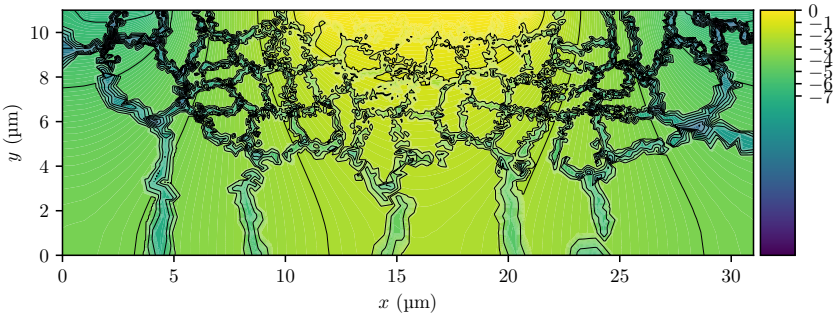


(b) After 50% of runtime

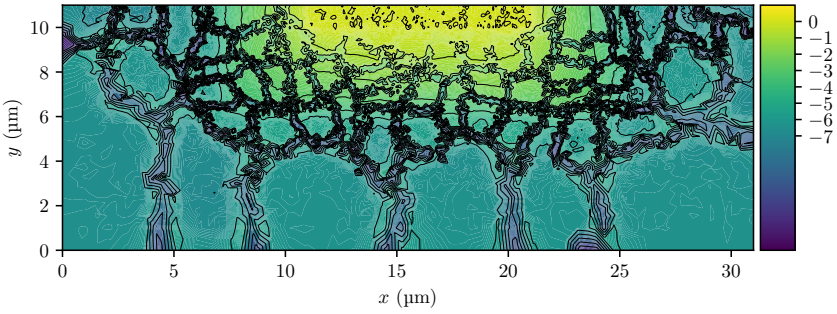


(c) At end of simulation

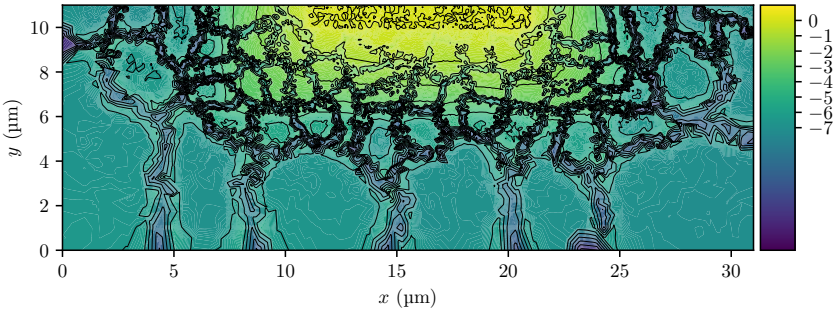
Figure 8.2: The electric potential at the start, in the middle and at the end of the simulation for 4 partitions. The scale at the left is the electric potential in volts.



(a) Start of simulation

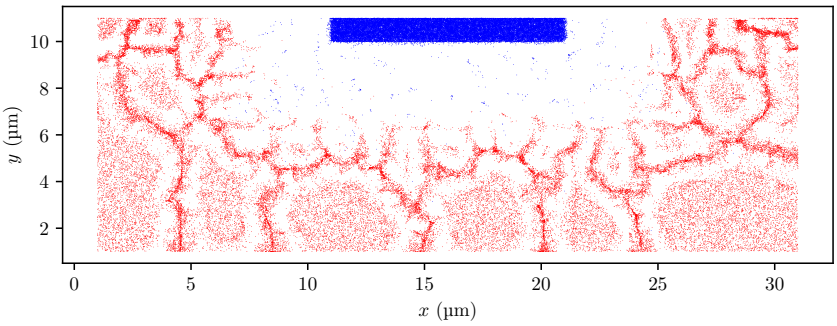


(b) After 50% of runtime

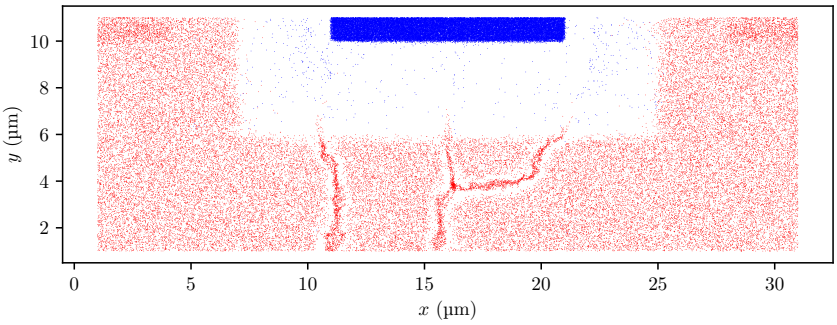


(c) At end of simulation

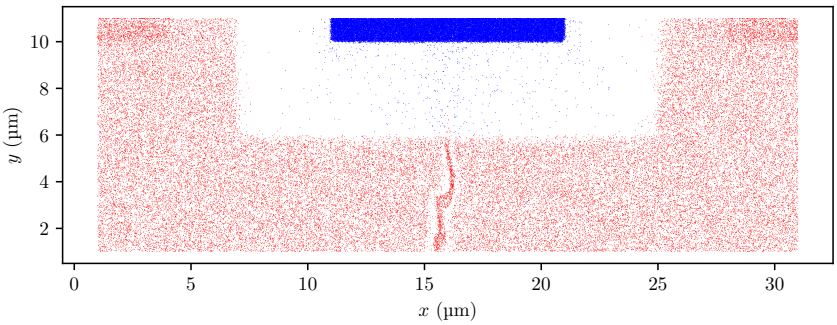
Figure 8.3: The electric potential at the start, in the middle and at the end of the simulation for 64 partitions. The scale at the left is the electric potential in volts.



(a) 64 partitions



(b) 4 partitions



(c) 2 partitions

Figure 8.4: The charge carriers of the system after the simulation has finished for different partition counts. Holes are coloured red and electrons are coloured blue. Notice how the holes seem to get trapped at the interfaces between partitions.

8.2 Runtime

Even though the results produced by the solver were not correct, a benchmark of its performance regardless of the solutions produced was attempted. Unfortunately the solver as it stands right now has a communication pattern which is too inefficient, leading to increased runtime for an increased core count. The communication patterns between processes at the moment has a lot of blocking operations and processes waiting for one another. In addition, vectors are communicated one element at a time, instead of using vectorized communications. The reasoning behind this was to first make sure the solver produced correct results before optimizing, but as the solver never produced any correct results, this optimization was never performed.

Chapter 9

Further work

The particle scattering routines of FFI-MCS are now parallelized, and there is a preliminary implementation of the parallel Poisson solver. There still exists many areas of the FFI-MCS which could still be improved, though. In this chapter some recommendations on future development of FFI-MCS are presented.

9.1 Fix the parallel Poisson solver

The current state of the parallel Poisson solver leaves a lot to be desired. While some work has been done to implement the parallelized preconditioned conjugate gradient method, it does not as of yet yield correct results.

This work is of the highest priority. The Poisson solver is one of the two integral parts of the simulation software, and if it does not produce correct results, the simulation will break down. Work should be done to see if the current implementation of partitioning and parallelization can be fixed, or if the solver has to be completely rewritten using another approach.

9.2 Optimise MPI communication patterns

As of now, the communication pattern in the parallel solver procedure is quite inefficient. Accumulation of a distributed vector is done by sending and receiving individual values of the distributed vector using blocking MPI procedures.

One approach could be to assemble all values belonging to one Gmsh partitioned entity[1], and then send and receive all values for that entity using a single MPI procedure. This would cut down the communication overhead drastically.

[1] A Gmsh parititioned entity can belong to several partitions.

Another approach is to use non-blocking MPI procedures. Right now, only half of the MPI ranks will be sending their distributed vector values at any one time. Using non-blocking MPI communications, one can cut down on the time wasted by waiting for other MPI ranks to send/receive.

Ultimately both of these improvements should be implemented into FFI-MCS. Neither is particularly challenging for someone familiar with distributed-memory parallelization and MPI, and so they would yield a good performance increase for relatively little work.

9.3 Migrate scattering routines from OpenMP to MPI

Right now, the particle scattering routines of FFI-MCS are using OpenMP for parallelization. The consequence of this is that we have two parallelisation techniques in FFI-MCS, OpenMP and MPI. Using MPI, one could probably improve the performance of the scattering routines by moving away from using shared memory, to having every MPI rank take care of a set fraction of the total superparticles. This way FFI-MCS would also become more scalable, and both the scattering routines and the Poisson solver could run on multiple nodes on a computing cluster.

9.4 Other parallelization areas

The speedup associated with an increasing process count in a parallel program is limited by the fraction of the program which is still serial. After the particle scattering routine and the Poisson solver have been parallelized and optimised, it is worth to look into whether other parts of the simulator can be parallelized to reduce the serial fraction of the program. This may for example be the electric field interpolation routine and routine for finding the particles' element positions.

9.5 Improving documentation

As of today, there is no proper documentation for FFI-MCS. There are project and master's theses written about its development, some including user guides, but it should not be necessary to read all of these to know how to use the program. In addition, the quality of the documentation of the code is very varying.

Another consideration is adding an automatic documentation generator to the project. There are different alternatives available, such as f90doc[2], Doxygen[3] and

[2]https://erikdemaine.org/software/f90doc/
[3]https://www.doxygen.nl/

FORD[4]. FORD seems like a good choice, as it generates nice-looking modern documentation pages, while handling Fortran source code better Doxygen, which is a popular documentation generator for many other programming languages.

9.6 Configuration and data files layout

Right now, configuration files for each simulation case are stored in their respective subfolders under the cases folder, while the output data is always put into the data folder. There might be merit in looking into other ways of structuring the file layout, such as outputting each case' output data into a subfolder in their respective case folder. This should however not be a prioritised improvement, as the current structure does work.

9.7 Version Control

FFI-MCS development has been passed on from developer to developer by means of just copying the project from one computer to another. Though one might be hesitant to turn to something newer and more complex, the benefits to version control tools like Git are not to be understated. Having a single repository for the code where new functionality can be developed in branches and releases can be tagged and stored in a controlled manner would be of great help to future developers.

9.8 Persistent distributed stiffness matrix

As of now, no routines have been implemented to save the distributed stiffness matrix to disk, which means that this has to be reassembled every time FFI-MCS is run. Similar to how the serial version of FFI-MCS works, it should be saved to disk, so that new simulations on the same mesh can be run without needing to reassemble the matrix.

9.9 Better output

The output from FFI-MCS at the moment can be very messy. There is a lot of debugging text which probably should be disabled by default. A config option, or a command line flag should be considered to enable or disable verbose logging. In addition, one can consider showing an estimate of how long the simulation has left.

[4] https://github.com/Fortran-FOSS-Programmers/ford

Bibliography

[1] Andreas Bolstad. 'Optimization of a Particle-Based Semiconductor Device Simulator: Self-Consistent Ensemble Monte Carlo Method'. Project thesis. Norwegian University of Science and Technology, 2020.

[2] Nicolai Stølen. 'Parallelization Strategy for the Poisson Solver of a 3D Monte Carlo Semiconductor Device Simulator'. Project thesis. Norwegian University of Science and Technology, 2022.

[3] Andreas Bolstad. 'Development of a 3D Particle-Based Device Simulator: A Self-Consistent Monte Carlo Approach Using Tetrahedral Grids'. Master's thesis. Norwegian University of Science and Technology, 2020. URL: https://hdl.handle.net/11250/2785543.

[4] Øyvind Olsen. 'Construction of a transport kernel for an ensemble Monte Carlo simulator'. Master's thesis. Norwegian University of Science and Technology, 2009. URL: http://hdl.handle.net/11250/246279.

[5] Ole Christian Norum. 'Monte Carlo simulation of semiconductors: Program Structure and Physical Phenomena'. Master's thesis. Norwegian University of Science and Technology, 2009. URL: http://hdl.handle.net/11250/246281.

[6] Øyvind Skåring. 'Ultrashort Relaxation Dynamics in Laser Excited Semiconductors'. Master's thesis. Norwegian University of Science and Technology, 2010.

[7] Camilla Nestande Kirkemo. 'Monte Carlo simulation of pn-junctions'. Master's thesis. Norwegian University of Science and Technology, 2011. URL: http://urn.nb.no/URN:NBN:no-29691.

[8] Aksel Jan Verne Vestby. 'Calculation of Terminal Currents in Single Photon Excited Avalanche Photodiodes'. Master's thesis. Norwegian University of Science and Tehnology, 2012. URL: http://hdl.handle.net/11250/246818.

[9] Ken Vidar Falch. 'Ensemble averaged and single Particle Auger Lifetimes in Zincblende Structure Semiconductors'. Master's thesis. Norwegian University of Science and Technology, 2013. URL: http://hdl.handle.net/11250/247121.

[10] Bjørnar Karlsen. 'Carrier Scattering Rates in Zincblende Structure Semiconductors derived from 14×14 k $\cdot$ p and ab initio Pseudopotential Methods'. Master's thesis. Norwegian University of Science and Technology, 2013. URL: http://hdl.handle.net/11250/247104.

[11] Tore Sivertsen Bergslid. 'Implementing a Full-Band Monte Carlo Model for Zincblende Structure Semiconductors'. Master's thesis. Norwegian University of Science and Technology, 2013. URL: http://hdl.handle.net/11250/247128.

[12] Juri Selvåg. 'High Precision, Full Potential Electronic Transport Simulator: Implementation and First Results'. Master's thesis. Norwegian University of Science and Technology, 2014. URL: http://hdl.handle.net/11250/247368.

[13] J J Harang. 'Implementation of Maxwell Equation Solver in Full-Band Monte Carlo Trans- port Simulators'. Project thesis. 2015.

[14] Theophile Jean Marie Chirac. 'Monte Carlo Simulation of Photoconductive Terahertz Sources in Mercury Cadmium Telluride'. Master's thesis. Norwegian University of Science and Technology, 2016. URL: http://hdl.handle.net/11250/2394152.

[15] David Kristian Åsen. 'Self-Force Reduced Finite Element Poisson Solvers for Monte Carlo Particle Transport Simulators'. Master's thesis. Norwegian University of Science and Technology, 2016. URL: http://hdl.handle.net/11250/2418022.

[16] Dara Goldar. 'Calculation of Wavefunction Overlaps in First Principles Electronic Structure Codes'. Master's thesis. Norwegian University of Science and Technology, 2017. URL: http://hdl.handle.net/11250/2461490.

[17] Mikael Haug. 'Schrödinger-Poisson and nonequilibrium Green's function methods applied to layered semiconductor devices'. Master's thesis. Norwegian University of Science and Technology, 2018. URL: http://hdl.handle.net/11250/2562316.

[18] Siri Narvestad Fatnes. 'A Three-Dimensional Finite Element Poisson Solver for Monte Carlo Particle Simulators'. Master's thesis. Norwegian University of Science and Technology, 2018. URL: http://hdl.handle.net/11250/2567223.

[19] Mats Estensen. 'Simulation of a Mercury Cadmium Telluride Avalanche Photo Diode'. Master's thesis. Norwegian University of Science and Technology, 2019. URL: http://hdl.handle.net/11250/2611911.

[20] Alfio Quarteroni. *Numerical Models for Differential Problems*. Vol. 16. MS&A. Cham: Springer International Publishing, 2017. DOI: 10.1007/978-3-319-49316-9.

[21] Susanne C. Brenner and L. Ridgway Scott. *The Mathematical Theory of Finite Element Methods*. Vol. 15. Texts in Applied Mathematics. New York, NY: Springer New York, 2008. DOI: 10.1007/978-0-387-75934-0.

[22] Giuseppe Pelosi, Roberto Coccioli and Stefano Selleri. *Quick finite elements for electromagnetic waves*. 2nd ed. Artech House electromagnetic analysis series. Boston: Artech House, 2009. 289 pp. ISBN: 9781596933453.

[23] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Second. Society for Industrial and Applied Mathematics, Jan. 2003. DOI: 10.1137/1.9780898718003.

[24] *OpenMP Application Programming Interface*. 2015. URL: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf (visited on 4th Feb. 2022).

[25] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. 9th June 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[26] Xiaohu Guo et al. 'Developing a scalable hybrid MPI/OpenMP unstructured finite element model'. In: *Computers & Fluids* 110 (Mar. 2015), pp. 227–234. DOI: 10.1016/j.compfluid.2014.09.007.

[27] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. Sept. 1998.

[28] Gennadiy Nikishkov. 'Basics of the Domain Decomposition Method for Finite Element Analysis'. In: (Jan. 2008).