The University of Manchester

UNIVERSITY OF MANCHESTER
School of Computer Science
Kilburn Building
Manchester M13 9PL

LARGE-SCALE FINITE DIFFERENCE TIME DOMAIN DATA PROCESSING USING HIGH PERFORMANCE SYSTEMS

Maksims Abaļenkovs

A dissertation submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Engineering and Physical Sciences

2007

to Dimitri

Abstract

Design verification of a Vivaldi antenna array has to be performed by the electromagnetic wave propagation in the time domain. The large-scale Frequency Dependent – Finite Difference Time Domain (FD-FDTD) numerical calculation is required for the evaluation of an antenna array structure. Modern numerical method implementations suffer from the inefficient I/O reducing the overall computation performace. The practical boundaries of certain supercomputer systems for the calculation of antenna elements are still unknown.

Therefore the current work addresses (i) the issues of the efficient storage and post-processing of data produced by the FD-FDTD simulation as well as (ii) testing the potential of two high performance computing systems for running the simulation code. The project introduces a new way for storing the data by means of the Hierarchical Data Format 5 (HDF5), which enables the parallelisation and drastically reduces the running time of data post-processing tasks. The estimation of maximum antenna elements feasible for efficient simulation on the IBM BlueGene/L (BGL) and Bull systems comprises another achievement of this work.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

September 7, 2007
Manchester

Maksims Abaļenkovs

Copyright

The Author

Being of Latvian nationality the author has started his higher education in the University of Latvia in the area of Mathematics and Statistics. After one year he changed his degree to Computer Science and continued his studies at Riga Technical University. There he gained a broad knowledge in a variety of engineering subjects as well as obtained a solid grounding in his major in Computer Science. In two years time he moved to Germany. He finished his undergraduate studies in Computer Science with a minor in Physics at the University of Düsseldorf. There he specialised in database systems and computer networks. Studying for Masters degree at the University of Manchester he became interested in the area of High Performance Computing and currently intends to pursue a PhD research in Electrical and Electronic Engineering.

Acknowledgements

The author would like to thank his supervisor Dr Fumie Costen for her enthusiasm, energy, guidance, motivation and faith, the HDF Group for their scientific data format and in particular Elena Pourmal and Barbara Jones for their support in the HDF5 installation process and library usage, Hopko Meijering for the introduction to the IBM BlueGene/L system and help on the job management procedure, Horace technical support team at Manchester Computing and especially Fiona Cook and Craig Lucas for their patience, guidance and willingness to help, João Costa for the explanation of FD-FDTD simulation software and raising interesting questions, Yongwei Zhang and Christos Argyropoulos for the discussion of an antenna functionality, Olga Kaiser for her mathematical and numerical expertise, Vadims Adamovičs for his advice on the software development and pseudocode review, Wentworth Miller for his inspirational intelligence, and parents for their interest and countenance.

Table of Contents

Total Word Count: 23610

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Chapter 1 introduces the area of computational electromagnetics. It defines the scope of the project and briefly describes the dissertation's structure. Afterwards follows a definition of an Ultra Wide Band. A section on applications of Ultra Wide Band concludes this chapter.

1.1 Project Scope

Simulation of Ultra Wide Band wave propagation under realistic conditions is a challenging task. Classical Maxwell's equations theoretically describe the effects of the electromagnetic wave dissemination. A number of numerical methods were developed for the calculation of Maxwell's equations for current industrial problems. Finding a practical solution requires solving a complex system of partial differential equations which puts high demands on the computational resources. High performance computing machines with large amounts of operating memory and processing power are usually used for faster calculation of these equations.

Classical Maxwell's equations describe the rise of electric and magnetic fields caused by static or moving electric charges. These equations provide a theoretical solution to a given problem. However, there is a strong need to obtain the numerical solution that could be used in practice. Different numerical methods were developed to approximate the theoretical solution to a practical one: Finite Difference Time Domain (FDTD), Frequency Dependent – Finite Difference Time

Domain (FD-FDTD), Alternating-Direction Implicit – Finite Difference Time Domain (ADI-FDTD), Finite Element Analysis (FEA) and many others.

The research group uses the FD-FDTD, an enhanced version of FDTD, which allows simulating the way a medium responds to the propagation of electromagnetic waves. There is a program written in Fortran. This code implements the simulation method and is capable of exploiting parallelism in the FD-FDTD algorithm. This enables running of the simulation on multiple computers and especially high performance systems.

The in-house software produces a number of output files, which represent the electromagnetic field values for a given time step of a simulation. The next stage of the simulation is the data post-processing. The temporal information contained in the output files should be extracted and transformed into spatial domain. The post-processing consists of two independent phases: (i) plotting of selected points within the simulation grid space and (ii) visualisation of wave propagation in a 2-dimensional plane – xy, yz or xz.

Initially the simulation data was stored in ASCII format and the post-processing tasks were performed by means of simple but highly inefficient bash scripts. This research focuses on the optimisation of the data post-processing methods. First, a number of scientific data formats were examined to provide an efficient storage and fast access to the simulation data. An HDF5 file format was selected because of its support for parallelism, high access speed and flexibility. During the project an output file structure using the HDF5 building blocks is designed and the in-house software is changed to produce the HDF5 output files. The next step is implementing the two phases of post-processing: point plotting and plane visualisation. A number of Fortran subroutines is introduced to accomplish these tasks.

Various performance tests on different hardware platforms are conducted to test the novel HDF5 code. A detailed comparison of the original ASCII/bash based post-processing with the new HDF5/Fortran/MPI based is made. The FD-FDTD simulation is usually performed for a high precision space and time grids and is run on super computers. The parallel power of high performance computing could be used for the data post-processing tasks as well. The performance experiments are run on the IBM BlueGene/L in Groningen (Netherlands) and on Bull system in Manchester (UK). Especially interesting is the comparison of these two machines because they have different architecture and hardware components.

The results of the tests also highlight the possibility for further improvement and optimisation of data production and post-processing tasks.

1.2 Project Structure

The structure of the dissertation is as follows. The next sections will describe the nature of the Ultra Wide Band and the areas of its application. Chapter 2 introduces the Maxwell's equations and after a short discussion on important numerical properties gives an overview and mathematical fundamentals of two numerical schemes important for current work – FDTD and FD-FDTD. The data post-processing problem is identified and analysed in detail in Chapter 3. Current situation in the research area of scientific data processing is presented in Chapter 4. It also contains a broad description of modern data formats used in scientific domain. The final format choice is justified by format performance benchmarks in Section 4.3.

Chapter 5 contains a thorough description of the project implementation. It introduces the new output file structure and the design of a novel algorithm for point plotting. Very interesting is the Section 5.3 which discusses possible code parallelisation approaches and gives the reasoning behind them. Chapter 6 contains the description of high performance computing systems used within this project. A broad performance testing of the super computers and estimation of maximum amount of Vivaldi antenna elements available for efficient simulation on these systems could be found in this chapter. Finally, Chapter 7 concludes the thesis with plans for the future work on overall project improvement.

1.3 Ultra Wide Band (UWB)

1.3.1 Definition

Ultra Wide Band (UWB) is any signal occupying width of 500 MHz or more in 3.1 to 10.3 GHz spectrum region. UWB signals obtain a number of interesting characteristics. Signals in this region use high-speed narrow pulses for data transmission. There is a relatively low interference level between communication parties

sharing the UWB spectrum. This allows to build UWB systems using simple hardware.

1.3.2 Application of UWB

The applications of FDTD algorithm exist for a very wide frequency range with frequencies varying from the very low to the optical level values. The circuit simulation, waveguide propagation, modelling of antennas and antenna arrays, studies of electromagnetic compatibility and pulse effects all are based on the FDTD. Combination of Maxwell's equations together with heat and energy transport equations plugged into FDTD helps to study bioelectromagnetic problems. These include hyperthermia, cancer detection, bioeffects of electromagnetic waves and Specific Absorption Rate (SAR) calculations. Engineering of *metamaterials*, materials which exhibit exotic electromagnetic properties not found in natural materials and compounds, is another application field of the FDTD method.

This project focuses on the application of FD-FDTD method for the calculation of UWB propagation. The main program used in the current work is aimed for the calculation of electromagnetic wave effects in the vicinity of a single Vivaldi antenna and Vivaldi antenna arrays.

UWB simulation is applied in waveguide[1] and Photonic Band Gap (PBG)[2] design. Initially the shape and material of these structures have to be known. The UWB is used to measure the electromagnetic field distribution and resonant frequencies inside of a waveguide. Correct measurement of these quantities is vital in future waveguide's construction. Figure 1.1(a) depicts a basic waveguide[3]. In case of PBG the electromagnetic wave simulation helps to measure the transmission and refletion coefficients on the surface and inwards of the structure. Figure 1.1(b) shows a PGB built of aluminium rods[4].

Vivaldi antenna[5] design and validation is another area of UWB research application. Analysis of radiation pattern around the antenna is conducted by means of

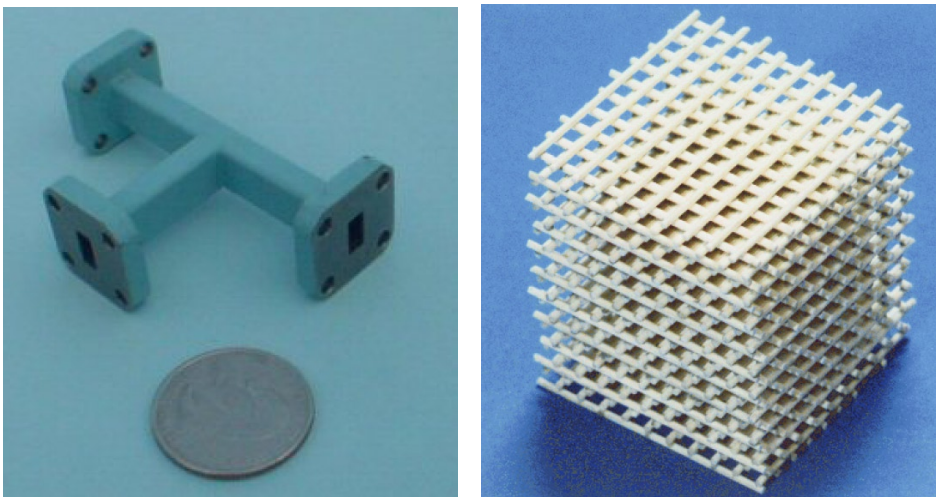
[1] Waveguide – is an artificial channel for guided dissemination of waves.

[2] Photonic Band Gap (PBG) – is a periodic optical structure used to influence the motion of photons.

[3] `www.microwaves101.com/encyclopedia/images/Waveguides`

[4] `www.public.iastate.edu/~cmpexp/groups/ho/pbg.html`

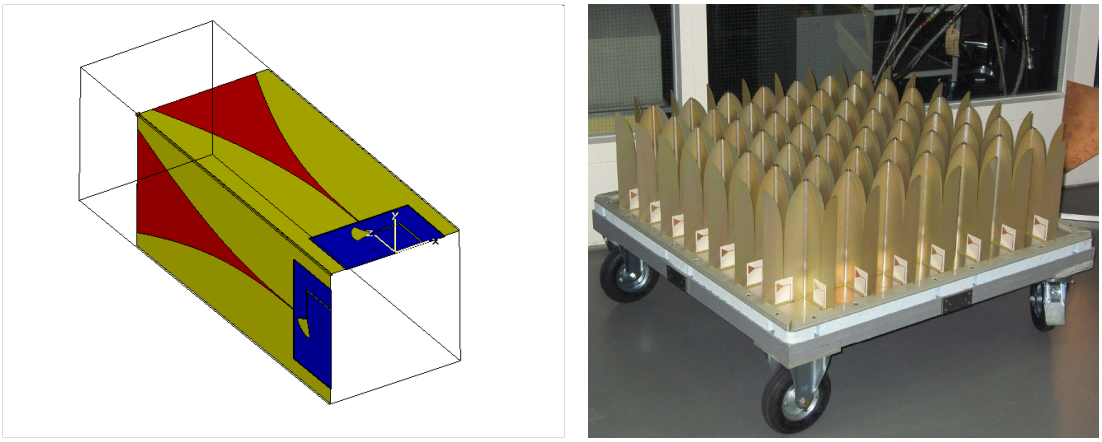
[5] Vivaldi antenna – is a special sort of antennas with improved directivity. Vivaldi antennas are best suited for transmission of broad spectrum signals.



(a) Basic Waveguide (b) Aluminium PBG Rod

Figure 1.1: UWB in Structure Design

FD-FDTD method and is important while constructing a new antenna element. Simulation of electromagnetic wave propagation for a Vivaldi antenna is one of the major interest areas of the research group. The main program used within this project calculates the wave emission of a Vivaldi antenna. Figure 1.2(a) is a schematical representation of a Vivaldi antenna [Zha07]. Figure 1.2(b) depicts a complete Vivaldi antenna array which consists of 97 elements[6].



(a) Vivaldi Antenna Scheme (b) Vivaldi Antenna Array

Figure 1.2: Vivaldi Antenna

UWB simulation is also used in medical research area. Measuring the electromagnetic field distribution in the vicinity of a human body helps to calculate the

[6] www.astron.nl

Specific Absorption Rate (SAR)[7] value. Figure 1.3 presents a visualisation frame from the simulation of a mobile phone wave emission[8].

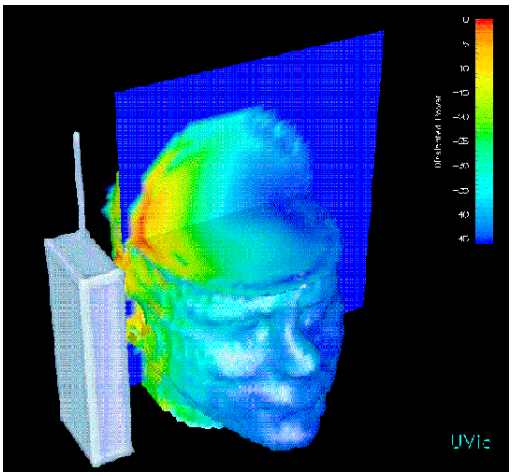


Figure 1.3: Radio Wave Propagation from a Mobile Phone

Medical image reconstruction could be done with help of UWB analysis. Different layers of human body are studied using the UWB systems. There are two sorts of simulation tasks known as *forward* and *backward* problems. The first stands for the emulation of UWB emission from a wave source, e.g. an antenna. The last corresponds to the analysis of electromagnetic field values at the receiver. Figure 1.4 gives a good example of a forward problem. It shows a conceptual schema of wave dissemination in a human brain.

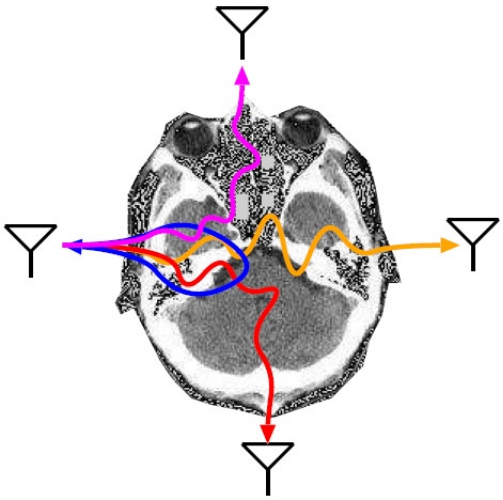


Figure 1.4: Sensing EM-wave Propagation in a Human Brain

[7]Specific Absorption Rate (SAR) – a measurement of a radio wave energy absorbed by the human body.

[8]`www.mobile-review.com/articles/2002/safety-en.shtml`

Main phases of a backward problem are depicted in Figure 1.5. A signal emitted from a wave source goes through a human body and returns to the receiver. The received signal processing is used for non-destructive image reconstruction, e.g. medical imaging based on microwave propagation is applied in cancer detection.

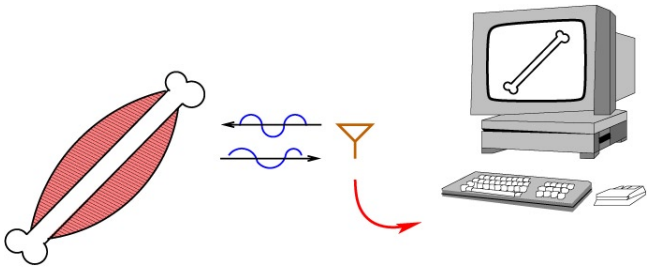
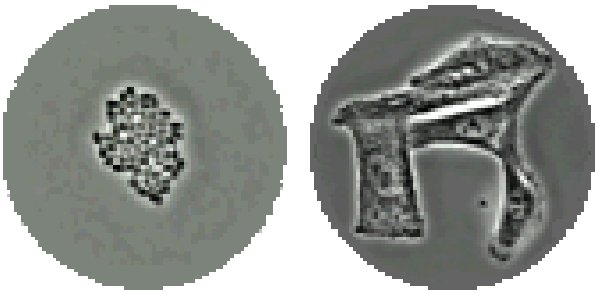


Figure 1.5: 3D Medical Image Reconstruction Using the UWB Radar

UWB is also applied in cell diagnostics research. The study of cell contents obtained by light pattern scattered from a cell helps identifying healthy and cancerous cells. Figure 1.6 gives an example of inner cell structure received by an UWB medical sensor[9].



(a) Healthy Cell (b) Cancerous Cell

Figure 1.6: Cell Diagnostics

1.4 Summary

A variety of numerical methods exists for the approximated solution of Maxwell's equations. The FD-FDTD scheme is capable of electromagnetic simulation including the medium characteristics. UWB is any signal broader or equal to 500 MHz in the 3.1 to 10.3 GHz frequency range. The UWB simulation is applied in many different areas such as waveguide and PBG structure design, Vivaldi antenna

[9] www.cellsalive.com

element and array construction, SAR calculation and cancer detection. High specification supercomputers are usually involved in running FDTD simulations. Current research focuses on the efficiency of the FD-FDTD data production and post-processing tasks during the simulation of wave propagation around Vivaldi antennas.

Chapter 2

Numerical Methods

Numerical method is a mathematical scheme specially designed for finding practical solutions to theoretical problems. A numerical algorithm is often used to solve very complex problems. In this case a numerical method only approaches an ideal theoretical solution. A series of high order approximations and algebraic operations such as matrix and vector multiplication frequently comprise a numerical algorithm.

2.1 Maxwell's Equations

Maxwell's equations define the character of electric and magnetic fields caused by static or moving charges. The electromagnetic fields are able to self-maintain and could also exist without charges or currents. The Maxwell's equations are:

$$\nabla \cdot \vec{\mathcal{D}} = \rho \tag{2.1}$$

$$\nabla \cdot \vec{\mathcal{B}} = 0 \tag{2.2}$$

$$-\nabla \times \vec{\mathcal{E}} = \partial_t \vec{\mathcal{B}} \tag{2.3}$$

$$\nabla \times \vec{\mathcal{H}} = \vec{\mathcal{J}} + \partial_t \vec{\mathcal{D}} \tag{2.4}$$

$$\vec{\mathcal{D}} = \varepsilon \vec{\mathcal{E}} \tag{2.5}$$

$$\vec{\mathcal{B}} = \mu \vec{\mathcal{H}} \tag{2.6}$$

where

ρ – volume charge density,
$\vec{\mathcal{J}}$ – volume current density,
$\vec{\mathcal{D}}$ – displacement vector,
$\vec{\mathcal{E}}$ – electric field vector,
$\vec{\mathcal{B}}$ – magnetic flux density vector,
$\vec{\mathcal{H}}$ – magnetic field vector,
ε – medium permittivity,
μ – medium permeability

All four vector field magnitudes $\vec{\mathcal{D}}$, $\vec{\mathcal{E}}$, $\vec{\mathcal{B}}$ and $\vec{\mathcal{H}}$ are functions of space and time $(\vec{r}, t)$. Equations 2.5 and 2.6 are called the *constitutive relationships*. These equations have the above form for the linear isotropic non-dispersive medium. Electromagnetic field in this type of medium always changes the same way independent of direction and a test particle's orientation. The constitutive relationships reflect properties of the medium, wherein the electromagnetic waves are propagating. Permittivity ε indicates the electric and permeability μ the magnetic properties of the medium. According to the values of ε and μ the medium influences the propagation character of the electromagnetic waves.

2.2 Important Numerical Properties

A good numerical approximation scheme for solving partial differential equations (PDEs) should obey the *Courant-Friedrichs-Lewy (CFL) condition* and three important mathematical properties. It should be *consistent*, *convergent* and *stable* [Cos05].

Courant-Friedrichs-Levy (CFL) Condition is a special convergence criterion for a PDE-solving algorithm. It sets the upper limit on a time step's size in an explicit numerical scheme. This means the discrete time step value used in the calculation should be less than a certain maximum. The CFL condition demands a time step to be smaller than the time a wave needs to cross the distance between two neighbouring space grid points. Equations 2.7 and 2.8 define the CFL condition for the general one-dimensional and three-dimensional cases respectively. The equations couple time step and grid

unit length values together. The numerical method will produce correct results when the CFL condition is maintained. The solution will diverge or it will not exist if the time step value exceeds the limit set by the equation.

$$\frac{u\Delta t}{\Delta r} < C \tag{2.7}$$

$$\frac{u_x\Delta t}{\Delta x} + \frac{u_y\Delta t}{\Delta y} + \frac{u_z\Delta t}{\Delta z} < C \tag{2.8}$$

where

u – electromagnetic wave velocity,
Δt – simulation time step,
Δr – length interval,
C – constant, depending on particular equation

Consistency is reached when the numerical method correctly approximates the analytical equations in case of progressive refinement of space-time lattice: $\Delta x, \Delta y, \Delta z, \Delta t \to 0$.

Convergence is a stricter property for a numerical scheme. In this case the numerical solution approaches the analytical, when the space-time lattice is refined at a given space-time point: $(x = i\Delta x,\ y = j\Delta y,\ z = k\Delta z,\ t = n\Delta t),\ \Delta x, \Delta y, \Delta z, \Delta t \to 0$ and $i, j, k, n \to 0$.

Stability is achieved when the difference of numerical solutions for arbitrary bounded differences in the initial values is limited in the same way as in the analytical equations. In general, stability indicates the algorithm's correctness in case of initial calculation disturbance such as rounding or approximation error. A stable algorithm will produce precise solution despite of the initial error. In contrary, the error can grow exponentially, widespread to the complete simulation grid and ruin the method's precision. Such method will be named unstable.

There is an active research going on in the area of unconditionally stable algorithm development. Decoupling of time-space interval dependency will allow relatively free selection of discretisation values. This will result in more efficient computation resource usage during the simulation. ADI-FDTD is one of the promising unconditionally stable numerical methods.

2.3 Finite Difference Time Domain (FDTD)

2.3.1 Description

The Finite Difference Time Domain (FDTD) is a powerful, yet simple method for the simulation of the electromagnetic phenomena. It is capable of calculating the propagation, radiation and scattering effects of the electromagnetic waves [GBOM04]. FDTD provides the solution of Maxwell's equations in the time domain. The method was developed by Kane S. Yee in 1966 [Yee66]. He was the first to apply the idea of finite differences to the problem of Maxwell's equations. The technique gained very high popularity in the scientific simulation domain. The classical FDTD method has survived a great number of enhancements and adaptations to various research areas. Currently it is the most known and widely used technique for the simulation of electromagnetic effects.

The general idea of the FDTD consists in replacing the derivatives in Maxwell's equations by finite differences. It is assumed, the values of the electromagnetic fields are known at a given time point. This starting position is called the *initial value problem*. The time dependent Maxwell's equations 2.3, 2.4 and the constitutive relationships 2.5, 2.6 build a system of hyperbolic partial differential equations. This system provides a unique solution of Maxwell's equations.

The FDTD defines an orthogonal cubic spatial grid, which is typically divided into smaller cubes called the *Yee unit cells*. Figure 2.1 depicts a typical Yee unit cell. This figure also shows the exact positions at which the electromagnetic values of E and H are calculated. If the material properties are important the permittivity ε and permeability μ values should be specified at every grid point.

The FDTD algorithm is explicit, which means that the current field values are functions of previous values in time. For the simulation of open problems the *Absorbing Boundary Conditions (ABCs)* are put in place to terminate the modeled planes. It must be said that the FDTD method is processor and memory intensive. Although the modern CPU and memory architectures are constantly improved, the growing complexity of simulation problems puts new demands on computing resources.

The second order finite centred approximation is applied to the time derivatives in Maxwell's equations. This approximation causes the numerical anisotropy and

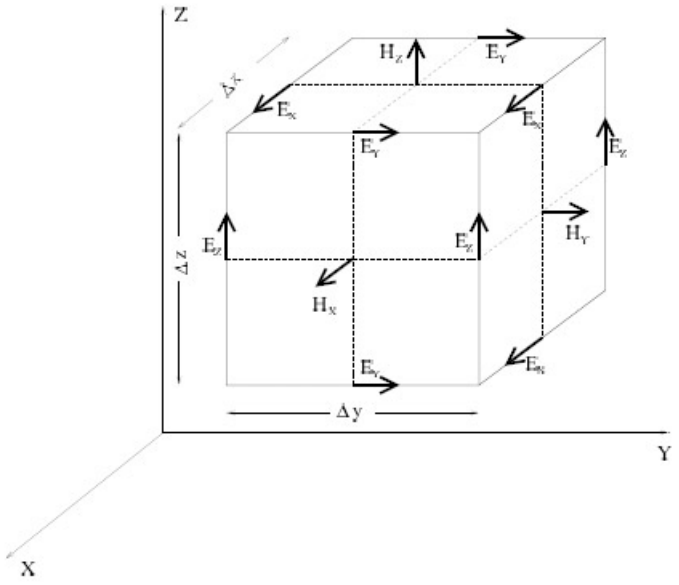


Figure 2.1: Yee Unit Cell

dispersion, which result in errors in the solution. These errors are negligible for electrically small problems. The low-phase techniques are used to reduce the errors in case of electrically large simulations. One of these techniques lies in using the higher order approximation to the derivatives.

Simulating materials with geometric inhomogeneities introduces another source of errors. Earlier a method called *staircasing*, was used to reflect the geometrical form of the modeled object. It tried to build a geometric structure of the object by means of the smallest grid unit, e.g. a Yee cell. Currently a hybridation of FDTD with FETD, MoMTD and other algorithms for the accurate simulation of geometric details is used. Instead of globally reducing the cell size of the grid to represent the geometric form of a model, modern simulation techniques use *sub-gridding* and *non-uniform meshing*. These approaches apply small-size cells only in places where the representation of fine geometrical details is needed and normal-size cells elsewhere.

The classical FDTD scheme follows the CFL condition, which states that the time increment's value depends on the space increment (see Section 2.2). The criterion also implies an upper limit to the time increment according to the space increment's value. This interdependency of time and space forces the simulation to use small time increments when a fine grid structure is modeled. And applying unnecessary small time increment values causes high CPU load during the simulation.

An extensive research in the FDTD field resulted in many research proposals being made. Some of these proposals focus on the better modelling of the material behaviour, whereas others study new methods of terminating the problem space. The *Perfectly Matched Layer (PML)* designed by Jean-Pierre Bérenger in 1994 [Bér94] is one of the widely adopted schemes for limiting the simulation grid space. This method's core idea is to truncate a Maxwellian medium with a non-Maxwellian. Where the non-Maxwellian medium perfectly absorbs all frequencies, polarisations and angles of incidence. This makes it a special sort of medium which completely attenuates all electromagnetic fields propagated within it. Only a few simulation steps are needed to exterminate any electromagnetic effects. Afterwards this absorbing layer is terminated by a perfect electromagnetic conducting wall (PEC/PMC). This last step eliminates the need for high computational resources in the simulation.

Nowadays a strong interdisciplinary exchange of ideas could be observed in various research fields applying FDTD. Suggestions from acoustics, computational fluid dynamics (CFD), quantum physics and many other subjects are being made to produce a more exact simulation algorithm. An important quality of the numerical simulation is its unconditional stability (refer to Section 2.2). A classical FDTD approach is stable only if the CFL criterion is maintained. New FDTD variations, which break the time-space interdependency, are being developed now [SS95]. *Alternating Direction Implicit FDTD (ADI-FDTD)* is one of the most promising unconditionally stable algorithms. It allows simulation of dense spatial grids with large time increments.

2.3.2 Mathematical Fundamentals

Starting with the assumption that currents are ohmic equations 2.9 and 2.10 may be used to define the electric and magnetic currents.

$$\vec{\mathcal{J}} = \sigma\vec{\mathcal{E}} \tag{2.9}$$

$$\vec{\mathcal{J}}^* = \sigma^*\vec{\mathcal{H}} \tag{2.10}$$

Here the magnetic current is taken for the symmetry reasons and symbols σ and σ^* stand for electric and magnetic conductivities. The constitutive relation-

ships 2.5 and 2.6 are true for the linear, isotropic and non-dispersive medium. This allows to omit the $\vec{\mathcal{D}}$ and $\vec{\mathcal{B}}$ in further calculations. Applying the ohmic current equations 2.9 and 2.10 to the time dependent Maxwell's equations 2.3 and 2.4 sets the Maxwell's equations into the E-H form:

$$-\tilde{\delta}_r\vec{\mathcal{E}} = \sigma^*\vec{\mathcal{H}} + \mu\partial_t\vec{\mathcal{H}}, \tag{2.11}$$

$$\tilde{\delta}_r\vec{\mathcal{H}} = \sigma\vec{\mathcal{E}} + \varepsilon\partial_t\vec{\mathcal{E}}, \tag{2.12}$$

$$\tilde{\delta}_r = \begin{pmatrix} 0 & -\partial_z & \partial_y \\ \partial_z & 0 & -\partial_x \\ -\partial_y & \partial_x & 0 \end{pmatrix} \tag{2.13}$$

Recalling the definitions for the centred difference 2.14 and centred average 2.15 operators it is possible to refer to Taylor series expansion, which states that $\delta_v f(v)$ is a second order approximation to $\partial_v f(v)$ and $a_v f(v)$ to the identity operation $\tilde{\mathcal{I}} f(v) = f(v)$.

$$\delta_v f(v) = \frac{f(v + \frac{\Delta v}{2}) - f(v - \frac{\Delta v}{2})}{\Delta v}, \tag{2.14}$$

$$a_v f(v) = \frac{f(v + \frac{\Delta v}{2}) + f(v - \frac{\Delta v}{2})}{2} \tag{2.15}$$

The FDTD method involves space and time discretisations. The space points are uniformly distributed at integer and semi-integer multiples of Δx, Δy and Δz values in each direction of the grid. Similarly, the time is discretised at integer and semi-integer multiples of Δt value. Then the electromagnetic field's value at any spatial-temporal localion could be described by the following universal notation:

$$\Psi^n_{i,j,k} \equiv \Psi(x = i\Delta x,\, y = j\Delta y,\, z = k\Delta z,\, t = n\Delta t) \tag{2.16}$$

Now the derivatives in the E-H Maxwell's equations 2.11 and 2.12 should be replaced with the centred difference operator δ_r and the conductivity averaged with a_t. These substitution obtains the classical FDTD method's equations in the general form:

$$-\tilde{\delta}_r \vec{E}^n_{i,j,k} = \sigma^*_{i,j,k} a_t \vec{E}^n_{i,j,k} + \mu_{i,j,k} \delta_t \vec{H}^n_{i,j,k} \tag{2.17}$$

$$\tilde{\delta}_r \vec{H}^n_{i,j,k} = \sigma_{i,j,k} a_t \vec{E}^n_{i,j,k} + \varepsilon_{i,j,k} \delta_t \vec{E}^n_{i,j,k} \tag{2.18}$$

$$\tilde{\delta}_r = \begin{pmatrix} 0 & -\delta_z & \delta_y \\ \delta_z & 0 & -\delta_x \\ -\delta_y & \delta_x & 0 \end{pmatrix} \tag{2.19}$$

Here the numerical curl operator $\tilde{\delta}_r$ (2.19) approximates the differentiation operator $\tilde{\partial}_r$ (2.13). The media heterogeneity in the FDTD space is obtained by sampling the analytical constitutive parameters ε, μ, σ and σ^* at every grid point:

$$\varepsilon_{i,j,k} = \varepsilon(x = i\Delta x,\, y = i\Delta y,\, z = i\Delta z) \ldots \tag{2.20}$$

Notice that 2.17 and 2.18 do not involve all six electromagnetic field components defined at every space point sampled with $\frac{\Delta x}{2}$, $\frac{\Delta y}{2}$ and $\frac{\Delta z}{2}$ values. Using only the field values specified in the Yee unit cell (refer to Figure2.1) allows to calculate 6 times coarser space grid.

Finally, taking the generalised FDTD equations 2.17 and 2.18, and sampling the electric field values at the semi-integer multiples of the time increment Δt and magnetic field values at the integer multiples will result in the marching-on-in-time algorithm:

$$\vec{E}^{n+1/2}_{i,j,k} = \frac{\varepsilon_{i,j,k} - \sigma_{i,j,k}\Delta t/2}{\varepsilon_{i,j,k} + \sigma_{i,j,k}\Delta t/2} \vec{E}^{n-1/2}_{i,j,k} + \frac{\Delta t}{\varepsilon_{i,j,k} + \sigma_{i,j,k}\Delta t/2} \tilde{\delta}_r \vec{H}^n_{i,j,k} \tag{2.21}$$

$$\vec{H}^{n+1}_{i,j,k} = \frac{\mu_{i,j,k} - \sigma^*_{i,j,k}\Delta t/2}{\mu_{i,j,k} + \sigma^*_{i,j,k}\Delta t/2} \vec{H}^n_{i,j,k} - \frac{\Delta t}{\mu_{i,j,k} + \sigma^*_{i,j,k}\Delta t/2} \tilde{\delta}_r \vec{E}^{n+1/2}_{i,j,k} \tag{2.22}$$

Here n represents the integer and i, j and k both integer and semi-integer values according to the field positions in the Yee unit cell on the Figure 2.1.

The classical FDTD scheme is self-maintained. Sampling the analytical initial conditions provides the numerical values for the start of the FDTD simulation. It must be said that the FDTD offers excellent opportunities for building higher or-

der schemes by applying other approximation techniques to the space and time derivatives ∂_r and ∂_t. There exist successful FDTD reformulations for other orthogonal coordinate systems such as polar and cylindric, and also general curvilinear coordinates.

2.4 Frequency Dependent – Finite Difference Time Domain (FD-FDTD)

2.4.1 Description

The Frequency Dependent – Finite Difference Time Domain (FD-FDTD) is another numerical method for the simulation of electromagnetic effects. Similarly to the classical FDTD it provides solution to Maxwell's equations in the time domain. As well as its predecessor FD-FDTD is an explicit algorithm and the ABCs should be specified to terminate the open problem space. The main difference between these methods is that the FD-FDTD variation reflects the simulation medium properties. This makes the FD-FDTD a more realistic approach for the electromagnetic simulations. However, this higher precision grade in handling the simulation medium comes at a price of increased calculation complexity. This involves more intense computing resource usage and longer processing time.

2.4.2 Mathematical Fundamentals

FD-FDTD considers the medium properties during the electromagnetic field simulation. To allow this the medium permittivity ε is set to be frequency dependent. Medium permeability μ stays constant. Permittivity is defined using the Debye model 2.23 and should be specified at every simulation grid point.

$$\varepsilon = \varepsilon_0 \varepsilon_r = \varepsilon_0 \left(\varepsilon_s + \frac{\varepsilon_s - \varepsilon_\infty}{1 + \jmath\omega\tau_D} - \jmath \frac{\sigma}{\omega \varepsilon_0} \right) \tag{2.23}$$

where

ε_r – relative medium permittivity,
ε_0 – free space permittivity,
ε_s – static relative dielectric constant,
ε_∞ – infinite relative dielectric constant,
τ_D – characteristic relaxation time of the dipole moment,
ω – angular frequency,
$\jmath$ – imaginary unit,
σ – medium conductivity

Applying the Debye model definition of the permittivity to 2.5 will result in the following equation for the displacement vector $\vec{D}$:

$$\vec{D} = \varepsilon_0 \left(\varepsilon_s + \frac{\varepsilon_s - \varepsilon_\infty}{1 + \jmath\omega\tau_D} - \jmath\frac{\sigma}{\omega\varepsilon_0} \right) \vec{E} \tag{2.24}$$

In addition to standard FDTD equations used to calculate the electromagnetic field values $\vec{E}$ and $\vec{H}$ (2.21, 2.22), FD-FDTD estimates the displacement vector $\vec{D}$ at each spatial-temporal increment of the computation. The $\vec{D}$, $\vec{E}$ and $\vec{H}$ values calculation advances in a circular fashion. Generally the calculation proceeds according to the following equations:

$$\frac{\partial \vec{D}^n}{\partial t} = \left[\vec{A}\right] \vec{H}^{n-1}, \quad \frac{\partial \vec{E}^n}{\partial t} = \left[\vec{B}\right] \vec{D}^n, \quad \frac{\partial \vec{H}^n}{\partial t} = \left[\vec{C}\right] \vec{E}^n \tag{2.25}$$

Where $\vec{A}$, $\vec{B}$ and $\vec{C}$ denote the spatial differentiation operators. Each of these variables represent a 3×3 matrix similar to 2.13. And $\vec{D}$, $\vec{E}$ and $\vec{H}$ are vectorised forms of displacement, electric and magnetic field values specified by Equations 2.26. The displacement value $\vec{D}^n$ is computed based on the magnetic field's value in the previous time step $\vec{H}^{n-1}$.

$$\vec{D} \triangleq \begin{pmatrix} D_x \\ D_y \\ D_z \end{pmatrix}, \quad \vec{E} \triangleq \begin{pmatrix} E_x \\ E_y \\ E_z \end{pmatrix}, \quad \vec{H} \triangleq \begin{pmatrix} H_x \\ H_y \\ H_z \end{pmatrix} \tag{2.26}$$

The FD-FDTD marching-on-in-time algorithm could be specified by the following general equation:

$$\vec{\phi}^{n} = (\tilde{\mathcal{I}} + \Delta t\vec{\psi})\vec{\phi}^{n-1} \tag{2.27}$$

Where $\vec{\phi}$ represents $\vec{D}, \vec{E}$ or $\vec{H}$ and $\vec{\psi}$ equals to $\vec{A}, \vec{B}$ or $\vec{C}$. This equation also highlights the explicit nature of the FD-FDTD method. It could be seen that the current value $\vec{\phi}^{n}$ is obtained using $\vec{\phi}$ from the previous time step.

2.5 Summary

Numerical method is a mathematical approximation of theoretical equations for providing practical solutions. Maxwell's equations specify the behaviour of electromagnetic fields in space and time domain. Constitutive relationships are important when the electromagnetic properties of the medium have to be considered in the simulation. These are omitted in the standard FDTD approach to obtain Maxwell's equations in the E-H form. Compliance with the CFL condition, consistency, convergence and stability are important mathematical properties of each numerical method. Both FDTD and FD-FDTD are conditionally stable and have interdependent space and time lattices.

Developed in 1966 FDTD still remains the most widely used algorithm for simulation of the electromagnetic wave propagation. Based on Taylor series expansion and centred difference and average operators it provides approximated solution of Maxwell's equations. FD-FDTD method takes into account frequency dependent electric properties of the simulation medium. Applying Debye model for the calculation of medium permittivity ε FD-FDTD allows to obtain more practical and precise simulation results but demands more computing and time resources. Current interchange and adaptation of ideas between different research areas stimulates the development of new FDTD variations and improvements.

Chapter 3

Problem Statement

This chapter gives an overview of the FD-FDTD data post-processing activities. It describes the initial status of data production and post-processing tools before the work undertaken in context of this project. Starting with the in-house software analysis for the FD-FDTD method implementation, it highlights the structure and character of the simulation output data. Section 3.3 comprises the core of this chapter providing a detailed dissection of point plotting and plane visualisation tasks. Section 3.3.3 contains the experimental and theoretical timing results for the data post-processing. Finally, the Section 3.4 discusses the performance measurements regarding the real-world processing requirements and justifies the need for more efficient parallel implementation.

3.1 In-House Software Analysis

The FD-FDTD numerical algorithm used in the UWB research work is implemented in Fortran programming language. Current simulation program includes a few C subroutine and directive calls as well as a number of MPI subroutines. The in-house software code is approximately 2,500 lines long. The C language interface is used for reading the grid space parameter data. Binary input file with an extention ".bin" specifies medium permittivity ε at each grid point. This bin-file is processed by four C methods `inittable()`, `tablelookup()`, `readnext()` and `finalizetable()`. Also there are `#define`, `#ifdef`, `#ifndef`, `#else` and `#endif` C compiler directives spread across the main code. These are applied for general program control. The original FD-FDTD algorithm implemen-

tation developed by Arnaud Thiry in context of his PhD dissertation [Thi06] was successfully parallelised by João Costa during his MSc research work [Cos06]. Native and user-defined MPI subroutines are used for FD-FDTD data scattering among a number of CPUs for simultaneous multiprocessing.

1: **set** $r_x, r_y, r_z, t_{max}, n_{CPUs} \leftarrow$ input parameter files
2: **initialise** simulation variables
3: **calculate** local r_z for each processor in $n_{CPUs} \rightarrow lr_z$
4: **distribute** $lr_z, n_{impulses}, spacedata$ between n_{CPUs}
5: **for** each time step $t \in [0, t_{max}]$ **do**
6: **for** each space point $p_{x,y,z} \in [r_x, r_y, lr_z]$ **do**
7: **calculate** $\vec{D}^n_{x,y,z},\ \vec{E}^n_{x,y,z},\ \vec{H}^n_{x,y,z}$
8: **calculate** Absorbing Boundary Condition
9: **store** $\vec{D}^n_{x,y,z} \rightarrow \vec{D}^{n-1}_{x,y,z},\ \vec{E}^n_{x,y,z} \rightarrow \vec{E}^{n-1}_{x,y,z}$
10: **output** $\vec{E}^n_{x,y,z},\ \vec{H}^n_{x,y,z} \rightarrow$ data files
11: **end for**
12: **end for**

Algorithm 3.1: FD-FDTD Method Implementation

Pseudocode in the Algorithm 3.1 shows the major steps of the FD-FDTD program. Variables r_x, r_y and r_z denote the x, y and z grid space coordinate ranges, whereas lr_z stands for local z range specified for a particular MPI processor. Parameters t_{max}, n_{CPUs} and $n_{impulses}$ define the maximum number of time steps, processors and impulses in the simulation. Medium-specific parameters read from a binary file are represented by a collective notion $spacedata$.

The FD-FDTD computation proceeds as follows. First the simulation parameters set in a number of input files are read and stored in local variables (line 1, Alg. 3.1). Afterwards all physical variables needed for the calculation are initialised (line 2). Then the simulation workload is distributed among the number of processors (line 4). Currently the processing work is divided according to the z axis. The local range of z value is calculated depending on an MPI rank of the processor. It will be stored into an lr_z variable (line 3).

After the MPI master processor with rank 0 distributes the local z range lr_z, $n_{impulses}$ and grid space data between all processors the program enters into the main simulation phase (lines 5-12). It consists of four loops: the first is for time step values and others for the grid space coordinates x, y and z. For the sake of algorithm clearness the three coordinate loops are represented with only one generalised cycle (line 6). The displacement $\vec{D}$ and electromagnetic field $\vec{E}$ and $\vec{H}$ values are computed in the main body of the loops (lines 7-10). After the Absorb-

ing Boundary Condition (ABC) calculation current $\vec{D}$ and $\vec{E}$ values are stored for the estimation of future $\vec{D}, \vec{E}$ and $\vec{H}$ parameters. Finally, the required simulation values are output into the data files.

3.2 Structure and Character of Produced Data

The FD-FDTD program produces one output file for each time step at each participating processor, e.g. simulation run for 100 time steps at 2 CPUs will create 100 time steps $\times$ 2 processors = 200 files. The file name is constructed according to the following template: `E_field_<time_step>_<rank>.out`. The first name's part `E_field` is constant and denotes the field values that were output in the current simulation, e.g. in this case only electric field E values. Other parts of the name `<time_step>` and `<rank>` show the time step and MPI processor rank of data kept in a particular file. All output data is stored in the ASCII format.

Space Coordinates			Field Values		
x	y	z	E_x	$\cdots$	H_z
1	1	90	-0.15396E+10	$\cdots$	-0.15041E+12
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
189	467	95	0.13878E+08	$\cdots$	0.13895E+10

Table 3.1: Output Data File Structure

Table 3.1 gives an example of a data file structure. The first three columns specify the grid space coordinates x, y and z. These are compulsory and always present in an output file. Afterwards come optional columns with the displacement $\vec{D}$ and electromagnetic field values $\vec{E}$ and $\vec{H}$. Maximally there could be up to 9 optional columns storing $\vec{D}_{x,y,z}, \vec{E}_{x,y,z}$ and $\vec{H}_{x,y,z}$ values. It is possible to set the output field values and coordinate ranges at the beginning of the main program, e.g. the simulation grid space ranges could be $x_s, y_s, z_s \in [0, 100]$ and output ranges $x_o, y_o, z_o \in [50, 100]$, whereas only $\vec{E}_z$ and $\vec{H}_z$ field values will be stored in data files.

Output of all displacement $\vec{D}_{x,y,z}$ and all electromagnetic field values $\vec{E}_{x,y,z}$ and $\vec{H}_{x,y,z}$ for the entire grid space will is required for the detailed analysis of the UWB systems. In this case each file will include 12 columns containing the most complete information about the simulation. On the other hand this storage pattern

will drastically increase the single file's output time and size.

3.3 Data Post-Processing Tasks

3.3.1 Plot Production

One of the most common data post-processing tasks is the plot production for a specific space grid point. This is usually needed for the verification of simulation results. All FD-FDTD data kept in output files is of a spatial character. Recall that a single file contains the displacement $\vec{D}$ and electromagnetic field $\vec{E}$ and $\vec{H}$ values for the entire grid space. But each file describes the status of a grid space for a particular time step only. The spatial data has to be transformed into temporal to construct a plot. This means all simulation files have to be traced for a given grid space point. Pseudocode in the Algorithm 3.2 explains the process of plot production.

1: **pass** x, y, z, col parameters of a grid space point p to `gawk`
2: **pass** data files F to `gawk`
3: **for** each file $f \in F$ **do**
4: **use** `gawk` to find $p_{x,y,z,col}$ in f
5: **output** file index f_i and field value $v_{col} \rightarrow$ `plot.data`
6: **end for**

Algorithm 3.2: Plot Production

Parameters x, y and z are the cartesian coordinates which define the spatial position of a grid point p. Variable col sets the target column number, i.e. the file column from where the field values will be read. For example, if an output file contains data for all electromagnetic fields, setting the target column number to 4 will result in reading the $\vec{E}_x$ values. A number of simulation data files is represented with the variable F, whereas f is a single output file and f_i denotes an ongoing file's index. Variable v_{col} identifies the electromagnetic field's value in a given column. The post-processing results are stored into a file called `plot.data`.

The algorithm proceeds in the following manner. All parameters required for plotting: point coordinates x, y, z, target column number col and a set of output files F are passed to the gawk[1] programming language. Afterwards each simulation file f is processed by gawk in search for a point p with a specified position

[1] gawk – is a special-purpose programming language aimed for flexible manipulation of text

x, y, z. The file index f_i and field value from a given column v_{col} will be extracted and saved in `plot.data` if such point exists in the file f. The algorithm terminates when all the simulation files are successfully processed. The resulting `plot.data` file will contain the required field's values for a static spatial position, but for all time steps. While the file index f_i represents the changing time step's value. This way an electromagnetic behaviour of particular space point could be tracked throughout the entire FD-FDTD simulation.

The point plotting algorithm is implemented with help of a `plotPoint.sh` bash[2] script. This script requests five command-line arguments:

```
plotPoint.sh <x> <y> <z> <col> <F>
```

For example a command launched with the following parameters:

```
plotPoint.sh 166 13 95 4 E_field_*_0007.out
```

will produce an output file with the $\vec{E}_z$ values for a point p with $x = 166, y = 13$ and $z = 95$. This will be true under assumption that only $\vec{E}_z$ field values were output during the simulation and all the space points with $z = 95$ were processed by a CPU with an MPI rank 7.

Time Step	Field Value
f_i	E_z
1	-1048348.
2	-2929066.
3	-7903632.
⋮	⋮
200	-2384146.

Table 3.2: Plot File Structure

Table 3.2 depicts the plot file `plot.data` structure. In this case the `plotPoint.sh` script was executed with the arguments shown above. The initial FD-FDTD simulation was run for 200 time steps. Finally, the `plot.data` file could be plotted by means of a Gnuplot[3] utility. Figure 3.1 shows the result of drawing the data

files. gawk is a GNU implementation of Unix programming language *awk* developed by Al Aho, Peter Weinberger and Brian Kernighan.

[2]bash (Bourne-again shell) – is a GNU shell or a command-line interpreter created in 1987 by Brian Fox. bash is an sh-compatible shell variation, which supports main Korn (ksh) and C shell (csh) functions.

[3]Gnuplot – is a portable command-line utility for plotting data and functions in two- or three-dimensional space. The project started in 1986 and is still under active development.

file `plot.data`. In this case the time step value is measured in Δt units, where $\Delta t = 1.924 \cdot 10^{-12}$ seconds.

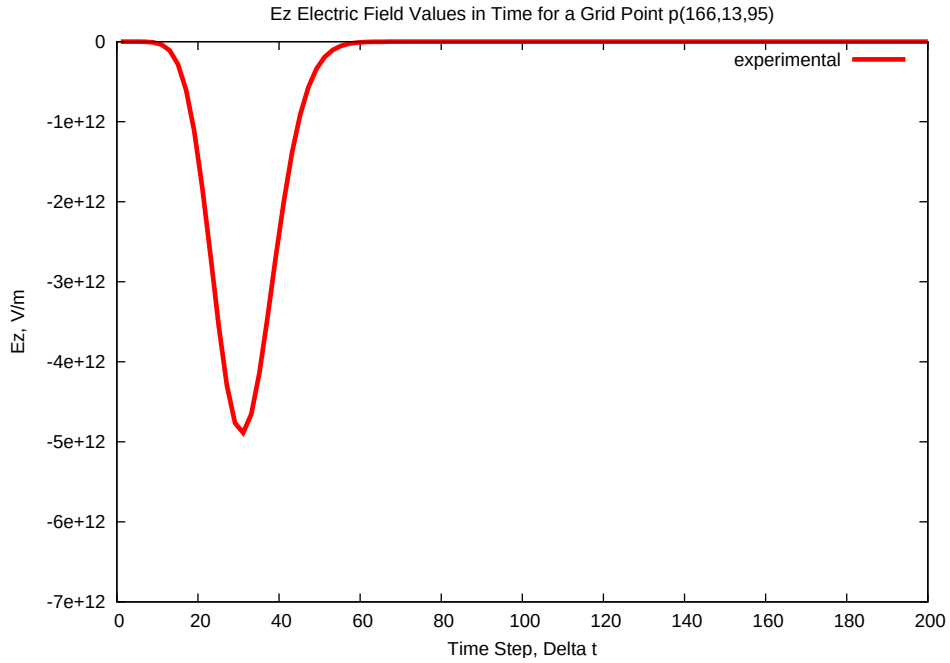


Figure 3.1: E_z Electric Field Values in Time for a Grid Point $p(166, 13, 95)$

3.3.2 Visualisation Production

Visualisation production is another frequently used post-processing task. Visualisation is needed for proofing the correctness of the FD-FDTD simulation. In contrary to the static plot, visualisation offers an animated representation of signal propagation along one of the simulation planes. Similar to the plot production the visualisation demands the FD-FDTD data to be transformed from spatial into temporal domain. Plotting the $\vec{D}, \vec{E}$ or $\vec{H}$ values in time is merely a collection of points connected into a line on the graph.

Whereas plotting requires only two parameters: time step and field value, visualisation occurs in a three-dimensional space and needs five parameters: time step, x, y, z space point coordinates and field value. Currently one of the space point coordinates is kept constant and the wave propagation is performed in a two-dimensional plane. The strength of the electromagnetic signal in the animation is presented with colour: red being the most intensive, green – moderate and blue – the weakest. Pseudocode in the Algorithm 3.3 shows the general steps of visualisation production.

1: **pass** coordinate value v_{a_3}, processor rank r, data files F to `gawk`
2: **for** each file $f_r \in F$ **do**
3: **use** `gawk` to find plane l_{a_1,a_2} specified by v_{a_3}
4: **output** $e \rightarrow$ `file.pgm`
5: **for** each file column $c \in (c_{a_1}, c_{a_2}, c_e)$ **do**
6: **use** `sort` to find $v_{max,c}, v_{min,c}$
7: **calculate** $(v_{max,c} - v_{min,c} + 1) \rightarrow cg_c$
8: **if** $(cg_c > g_c)$ **then**
9: **replace** g_c with cg_c in `file.pgm`
10: **end if**
11: **end for**
12: **use** Fortran program to convert `file.pgm` $\rightarrow$ `file.ppm`
13: **end for**
14: **use** ImageMagick to convert `*.ppm` $\rightarrow$ `movie.mpeg`

Algorithm 3.3: Visualisation Production

Parameters a_1, a_2 and a_3 in the pseudocode denote the cartesian coordinate axes, e.g. $a_1 = x$, $a_2 = y$ and $a_3 = z$. Whereas a is a general notion for any possible axis. Variable v stands for a coordinate value. MPI processor rank is specified by the parameter r. A subset of FD-FDTD data files used in the process of visualisation production is set by variable F, while f_r stands for a particular file initially produced by a CPU with rank r. Parameter l with two indices written in subscript defines the target visualisation plane. Variable e represents a generic electromagnetic field's value $\vec{E}$ or $\vec{H}$. This is the actual field value aimed for animation. Notion c specifies the processed column in a file f_r. The target column could be either the coordinate axis a_1, a_2, a_3 or the electromagnetic field e. Parameters g_c and cg_c denote the value range in a particular file column c, whereas g_c is the value stored in the pgm-file and cg_c is the currently calculated value range for a particular file f_r.

The visualisation of signal propagation is produced in the following way. First, the coordinate value of one of the simulation dimensions v_{a_3} is passed to gawk programming language. Parameter v_{a_3} sets the target visualisation plane. In the case shown it will be the value in the z axis. This means the visualisation will be produced in the a_1-a_2 plane, i.e. the x-y plane. The processor rank r and the data files F are also submitted to gawk. Parameter r tells the gawk to search for the target information in those files in F, that were produced by the specified processor. Only those files will contain the data required.

Then each file f_r in the given subset F is traced by gawk. As soon as the pro-

gramming language finds the given plane l_{a_1,a_2}, the required electromagnetic field value e is output to the pgm-file.[4] Afterwards the program enters into a short loop to conduct a series of similar actions for each target column of the file f_r. First the coordinate axis a_1 and a_2 and then the electromangenic field value e column are traced. The `sort`[5] utility is applied to find the maximum and minimum values v_{max}, v_{min} of a column c. After the calculation of current value range cg_c, it is compared to the respective file value range g_c. If the current range cg_c appears to be larger, it replaces the g_c value in the pgm-file.

After the file f_r is successfully processed by gawk and `sort`, an in-house Fortran program called `pgm2ppm.f` is applied to the resulting pgm-file to convert it into a ppm-file[6] `file.ppm`. The last stage of the visualisation production is the PPM to MPEG file conversion, which is performed by means of an ImageMagick[7] `convert` utility. This step transforms the number of static signal representations in PPM format into an animated MPEG wave propagation `movie.mpeg`.

The C shell[8] script `visualisePlane.csh` implements the plane visualisation algorithm. This program requests two parameters `<plane>` and `<rank>`, which specify the desired visualisation plane and the MPI processor rank for the subset of target out-files. The command-line arguments `<plane>` and `<rank>` refer to the v_{a_3} and r variables in the Algorithm 3.3. The FD-FDTD simulation files F addressed by the program will be all out-files in the current directory. The script is usually launched as follows:

```
visualisePlane.csh <plane> <rank>
```

For example a command run with the given parameters:

```
visualisePlane.csh 95 0007
```

will animate the wave propagation in the x-y plane, where $z = 95$ and the CPU rank equals to 7. Tables 3.3 and 3.4 give an example of the PGM and PPM file

[4]PGM (Portable Grayscale Map) – is a lowest common denominator grayscale image format. It uses 8 bits to store 1 pixel of information.

[5]sort – is a standard Unix command-line utility for sorting files and data.

[6]PPM (Portable Pixel Map) – is a lowest common denominator colour image format. It uses 24 bits to store 1 pixel of information: 8 bits for each of the key colours – red, green and blue.

[7]ImageMagick – is a comprehensive set of command-line utilities for manipulation of bitmap images. Software implementations exist for all major operating systems as well as the interfaces supporting a wide variety of modern programming languages.

[8]C Shell (csh) – is a Unix shell with the syntax modelled after C programming language. It was developed by Bill Joy at the University of California at Berkeley.

contents. Both PGM and PPM formats used for the visualisation production task are the plain format versions. The PGM and PPM file structures are similar. First row occupies a "magic number", which identifies the file type. The next row is a comment preceded by a hash sign. Then follow the coordinate axis ranges g_{a_1}, g_{a_2}. The electromagnetic field value range g_e is stored afterwards. All other values in the file show the electromagnetic field's strength for the complete grid space. Figure 3.2 depicts a single PPM file for the time step $t = 150$. Every PPM file is one visualisation frame. Converted into MPEG format, the entire collection of PPM files comprises the animation of the wave propagation. There is no signal within the vertical and horizontal white lines on the image. This is because these lines represent the metal body of the actual Vivaldi antenna, which emits the signal.

P2
test
467 189
31
0
⋮
22
24
31
⋮
0

Table 3.3: PGM File Structure

P3		
# test		
467	189	
255		
0	0	0
⋮	⋮	⋮
199	255	55
131	255	123
99	255	155
⋮	⋮	⋮
0	0	0

Table 3.4: PPM File Structure

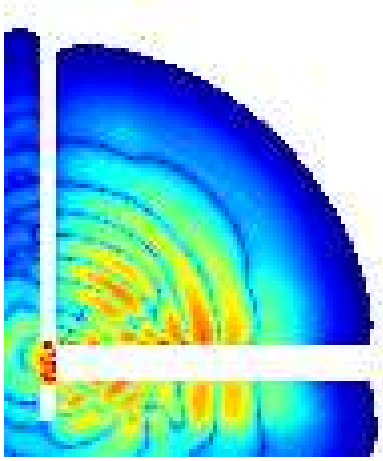


Figure 3.2: Visualisation Frame of the Wave Propagation for Time Step $t = 150$

3.3.3 Timing Results

A series of performance experiments for plot and visualisation production were run on three machines in the local laboratory: 25 and 36, 55. The research machines are named by the last numbers of their IP addresses. The detailed specifications of these test systems are given in the Table 3.5. System 55 is a laptop. The FD-FDTD simulation data used for post-processing tasks was also produced in the local cluster. This simulation was run for 200 time steps $t \in [0, 199]$, with the grid space of $189 \times 467 \times 5$, whereas $x \in [1, 189]$, $y \in [1, 467]$ and $z \in [96, 100]$. The resulting output file contained the x, y, z space coordinates as well as the electric field's E_z values. In total each simulation's out-file consisted of $189 \times 467 \times 5 = 441,315$ text lines and occupied approximately 10 MB of disk space.

Name	Operating System	Kernel Ver.	CPU	Core	Speed, GHz	RAM, GB	L2-Cache, KB
25	openSuSE 10.2 (i586)	2.6.18.8-0.1-def.	AMD Athlon XP 2000+	1	1.667	2.0	1×256
36	openSuSE 10.2 (X86-64)	2.6.18.8-0.1-def.	AMD Athlon 64 X2 4200+	2	2.200	4.0	2×512
55	openSuSE 10.2 (X86-64)	2.6.18.8-0.3-def.	Intel Core 2 Duo T5500	2	1.667	1.0	1×2048

Table 3.5: System Specifications

Table 3.6 summarises the timing results measured on the test systems. Table row "Columns" specifies the number of columns in the FD-FDTD output file: 4 – refers to $x, y, z, \vec{E}_z$; $9 - x, y, z, \vec{E}_{x,y,z}, \vec{H}_{x,y,z}$; $12 - x, y, z, \vec{D}_{x,y,z}, \vec{E}_{x,y,z}, \vec{H}_{x,y,z}$. The plot and visualisation time measurements for 4-column files are the only results obtained experimentally. The 4-column time values given in the Table 3.6 are the averaged timings of three independent experiments. All other measurement data (for 9- and 12-column files) was estimated theoretically based on the experimental

timings.

	Plot			Visualisation		
Columns	4	9	12	4	9	12
25	3:44	8:24	11:12	1:01:57	2:19:23	3:05:51
36	1:26	3:14	4:18	16:18	36:41	48:54
55	1:35	3:34	4:45	20:05	45:11	1:00:15

Table 3.6: Plot and Visualisation Production Experiments

Analysing the performance data it is possible to make a number of conclusions. The dual core 64-bit machine with an appropriate operating system gives almost a triple performance improvement: 1:26 compared to 3:44 for plot production with a 4-column output file. In case of visualisation production the performace difference rises to the quadruple timing results: 16:18 on a dual core against 1:01:57 on a single core machine. The CPU frequency and the cache size are vital for the speed of post-processing. However, the amount of available RAM is of a secondary importance. Even though the 36th machine has 4 times more operating memory – 4.0 GB compared to 1.0 GB on the 55th it was not able to drastically outperform the system 55.

Also there was an assumption that the 36th was producing the plots and visualisations faster because the FD-FDTD data was stored on the local disk. In contrary the 25th had to acquire the output files via a LAN. This assumption was disproved by another test on the 25th with the simulation data and post-processing scripts stored on the local disk area of the 25th. In that case the post-processing timing results on the 25 varied in terms of a few minutes. This allows to state that the network communication time during the data post-processing is negligible compared to the entire amount of time required to produce a plot or a visualisation.

3.4 Summary

The in-house software implementing the FD-FDTD numerical method is written in Fortran using C and MPI subroutines. The medium permittivity parameter ε specified for every space grid point is stored in the binary file `param00000.bin`. The FD-FDTD simulation produces the 4- to 12-column large data files in ASCII

format, whereas the first three columns contain the x, y, z grid space cartesian coordinates and the remaining optional columns specify the displacement $\vec{D}$ and electromagnetic field $\vec{E}, \vec{H}$ values. Both plot and visualisation production are required for the verification and better representation of the FD-FDTD simulation data. The post-processing tasks transform the spatial data character into temporal. While the plot shows a signal propagation for a fixed grid space point, the visualisation offers an animation of electromagnetic wave in a given plane. It also highlights the signal's strength with colour.

Although, the data post-processing on a high specification 64-bit dual core machine is three to four times faster than on a modest single core system, the performance gain is not enough for plotting and animating the real-world FD-FDTD data. The practical data post-processing usually requires the 9-column large files from the 10,000 time step long simulation. The file single file size in that case grows up to 1 GB. Considering the 200 time step measurements for the 9-column files on the high specification system still requires 3:14 and 36:41 for plot and visualisation production tasks.

This means storing the simulation data in the ASCII form is not efficient. There are alternative file formats developed specifically for scientific data storage. Furthermore, the data post-processing based on standard Unix implementations of gawk programming language and `sort` utility is not appropriate for large-scale FD-FDTD data processing. One of the major aims of this research work is the development of more efficient parallel data-postprocessing implementations, capable of running on High Performance Computing (HPC) systems.

Chapter 4

Scientific Data Processing

This chapter begins with discussion on the current status of the scientific data processing area. It also highlights the major development trends in the scientific data manipulation domain. Then the modern special-purpose data formats are described. Another section is focused on the format performance issues. Final thoughts on the format choice for this project and the justification of this decision conclude the chapter.

4.1 Current Situation

The character of contemporary scientific data determines the data structures used for data storage. Typical research data is usually kept in multidimensional arrays, tables of records and images. Among the major challenges the scientific computing faces nowadays are growing data complexity, tremendous data volumes, and the need for parallel, remote and distributed data processing. Metadata is one of the instruments for managing large amounts of data. The scientific community becomes aware of the need for metadata, the aspects of its usage and the benefits it brings to the research.

The scientific data volumes are doubling each year. There is a strong need for dataset integration, better data exploration and interactive analysis tools. The biggest gap on a way to improved analysis instruments currently exists in human-machine interface. Huge amounts of data tend to confuse a scientist and make the research more complicated. Scientist should be put back in control of his data

and be able to use his intuition to conduct successful experiments.

Modern scientific algorithms become more sophisticated and most of them are super-linear, which means N^2 or N^3 time is needed to process N data points. Low I/O bandwidth in comparison to storage capacity make the picture even worser. All of these facts result in a longer analysis time and state a clear demand for improved scientific data processing tools.

Jim Gray et al. in [GLNS+05] define a notion of a *smart notebook*, which stands for an easy and intuitive data management. The general idea behind this smart notebook is that huge computing centres take over storage of all scientific datasets and applications providing a scientist with a *personal workspace*. In this case a scientist will only communicate his data requests and receive computation results to and from the data centre. The data centres will be self-healing and serve for data loss protection and reduced configuration access.

Metadata is the data about raw data. One of its most important characteristics is that it stores information about the data lineage, the way it was computed, obtained and measured. Contemporary multimedia formats such as MP3, JPEG and PDF already offer this feature. The metadata is crucial for many aspects of smart notebook's development scenario. It contributes for better understanding of scientific data by people and analysis tools. It enables implementation of simple and convenient data models for organising and representing data arrays and relationships among them. Many popular scientific formats like HDF, NetCDF and FITS making use of metadata. The self-describing property of the data is critical for data independence. Independence means separating applications from the raw data. Similar to modern database systems a physical and logical data independence could be achieved by means of metadata. Data independence makes it possible to change underlying physical or logical data organisation without the need to change the processing applications.

Another trend which will ease the transition to the smart notebook world is the development and convergence of intra-disciplinary ontologies into a single global ontology. This gross ontology will simplify the design of data analysis tools.

Whereas current scientific data formats offer good means for storage and organisation of tabular numerical data, their searching and analysing abilities are rather limited. Usually a *filter-then-analyse* data processing approach is followed when using HDF, NetCDF and FITS, i.e. an application tends to process one file at a

time to find the data needed and pass it to the programming language for further modifications. Moreover, this method is frequently used with only a single CPU available for data processing.

The Message Passing Interface (MPI) might be a good alternative in this case. But it is only beneficial for highly regular tasks, and offers neither metadata nor indexing. Although, many scientific formats started to deploy MPI capabilities and improve their processing performance, there is still a clear demand for (i) intelligent indices and search subsetting, (ii) parallel processing and access to data and (iii) powerful analysis tools.

In the coming decade the field of data processing will concentrate its efforts on bringing together the main components of efficient data manipulation – file systems, database systems and programming languages. Historically scientific data has been organised in files. So the development of advanced file systems aimed for high performance data processing will be a part of the future research. On the other hand modern database systems already offer many features needed in scientific community. They have indexing, replication and distributed access capabilities. Current database systems driven by commercial organisations have standardised around the relational SQL model. Which is not always suitable for scientific applications.

Programming languages and especially object-oriented approach have always played an important role in research. Nowadays there is an *impedance mismatch* phenomenon, which stands for a mismatch between a relational model used by modern database systems and an object-oriented model operated by programming languages. The future of efficient data processing lies in the evolution and integration of file systems, database systems and programming languages. There are prototypes of object-oriented database systems and databases allowing storage of embedded of linked records. The scientific data files could be plugged into the database by means of a linked record mechanism. Moreover its awaited that the well-known scientific formats will be integrated into popular database systems as pre-defined datatypes.

The confluence of these major components will bring many benefits to the scientific community and result in better analysis tools. Among others this symbiosis will enable automatic data parallelism, indexing, non-procedural access and search capabilities, data independence, federation and flexible structuring, replication, backups, automated design and management.

Some attempts have been made to conduct scientific data analysis with help of database systems. Due to better indexing and parallelism it proved to be simpler and faster than the file-oriented approach. A survey described in [GLNS+05] highlights the main reasons of little acceptance of relational databases in research. First of all, the standard relational model does not support multi-dimensional data, coordinate systems and relationships among elements in data arrays well enough. Resource-expensive learning of new applications, absence of good plotting and visualisation tools, lack of support for specific datatypes and access patterns, low speed, need for an expensive database administrator and no ability to manipulate data with the standard applications after it was loaded into the database were named among major drawbacks of relational database systems.

Although, the popular scientific data formats still lack non-procedural query analysis, automatic parallelism and sophisticated indexing capabilities, the NetCDF, HDF, FITS and Google Map-Reduce are a first step towards the future data processing. These formats might be considered simple database systems because they provide schema language to define metadata, basic indexing strategies and elementary data manipulation language. They have given birth to parallel, non-procedural programming and allowed portability. These formats also provide a set of tools to create, access, search and visualise the data.

4.2 Data Formats

4.2.1 Network Common Data Form (NetCDF)

Network Common Data Form (NetCDF) is a data abstraction and a software library intended for working with multidimensional data. The data format is based on eXternal Data Representation (XDR) by Sun Microsystems and therefore provides machine-independency for storage of scientific data. In contrary to architecture-specific binary files, data kept in NetCDF files is self-describing, which broadens its inter-application usage potential. From the beginning the format was developed for an efficient access to small portions of data. Supporting the named multidimensional variabes, coordinate systems and named auxiliary properties the NetCDF data abstraction offers a high level of flexibility for scien-

tific dataset definition. A single hyperslab[1] access call is used by NetCDF instead of multiple single read and write operations. A hyperslab is identified by specifying a variable name, corner and edge vectors. NetCDF also implements a feature called an *unlimited dimension*, which is a special index that might be used for the record-oriented data access.

A NetCDF file consists of three basic components: *dimensions*, *variables* and *attributes*.

Dimension is a structure defining the shape of a dataset. This might be a real physical dimension, coordinate system or another abstraction.

Variable is an array of elements of the same datatype. It is a major construct for storing raw data. A NetCDF variable is specified by a list of dimensions and list of attributes.

Attribute provides additional information about variables and files.

A file is extendible, which makes adding new components to an existing dataset possible.

NetCDF stores all data in linear fashion. Internally the data is represented as a one dimensional array of variable size. This abstraction allows NetCDF to minimise the I/O overhead and reach good data processing performance. To optimise the data manipulation further the PnetCDF was developed, which is a parallel version of NetCDF.

There also exist `ncdump` and `ncgen` utilities capable of transforming NetCDF binary data into a readable text form and backwards. These utilities could be called within a C or Fortran code to create a NetCDF representation of a given text file.

Performance tests run by Rew and Davis [RD90] have shown that the CPU time for XDR conversion during the data manipulation is negligible in comparison to the disk access time. Main areas of this format's application are data archiving, dissemination and representation among distributed programs.

[1]Hyperslab – is a subset of a complete dataset. Hyperslab is a selection of contiguous points in the dataspace or blocks of points yielding to a regular pattern.

4.2.2 Hierarchical Data Format 5 (HDF5)

Hierarchical Data Format 5 (HDF5) is a relatively new file format and software library implemented in 2002. It was specially developed to address the properties of scientific data and make the operation on this data maximum efficient. HDF5 serves for scientific data storage, access, management, exchange and archiving. It is built upon a very flexible data model, provides system-independent file format and library, supports large and complex datasets and data processing in massively parallel environments [Fol02].

HDF5 does not set any limit on the data file size and the number of objects stored in a single file. There are many programming interfaces developed to enable HDF5 functionality on wide variety of computing platforms. These include C, C++, Java and Fortran 90 interfaces. The format has a versatile data model making it possible to express other formats in terms of HDF5.

Conceptually an HDF5 data file is a container. The file format operates with primary and secondary types of objects, the first consisting of *groups* and *datasets*, and the second holding *datatypes* and *dataspaces* [Gro07].

Group is a directory-like structure organising datasets and other groups.

Dataset is a multidimensional array of elements of any datatype.

Datatype defines the nature and character of data in the dataset.

Dataspace is an auxiliary entity keeping information on the shape and intended usage of data.

The HDF5 groups and datasets could be compared to Unix directories and files. There is a *root* group in each HDF5 file by default. Both dataset and group objects can have metadata, which consists of attributes of the form “name=value”. There also exist links between objects, which enable object sharing. Figure 4.1 represents a general concept of an HDF5 file.

HDF5 offers wide means for describing the data with common and user-defined metadata. The Virtual File Layer (VFL) abstraction enables efficient I/O operations, including standard, parallel and network I/O. HDF5 parallel data processing mechanism relies on MPI I/O, specified by the MPI-2 standard. The HDF5

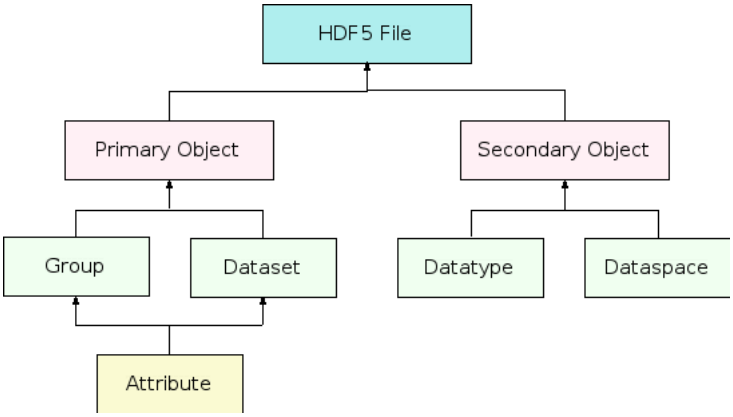


Figure 4.1: HDF5 Object Hierarchy

I/O library is dual-layered and might be used differently. On the high-level of abstraction it offers object-specific and on the low-level media-specific APIs. It also supports diverse storage media. HDF5 split driver allows to store the data in two physical files – one for meta- and the other for raw data.

Many strategies exist for optimising efficient storage and manipulation of data. The data could be compressed and chunked, and data structures extended and updated. HDF5 compresses data in GNU zlib by default, but it could be replaced by any other compression algorithm. Chunking strategy proves to be very efficient, especially in cases when partial data access is needed. Figure 4.2 depicts some commonly used data storage options [Gro02].

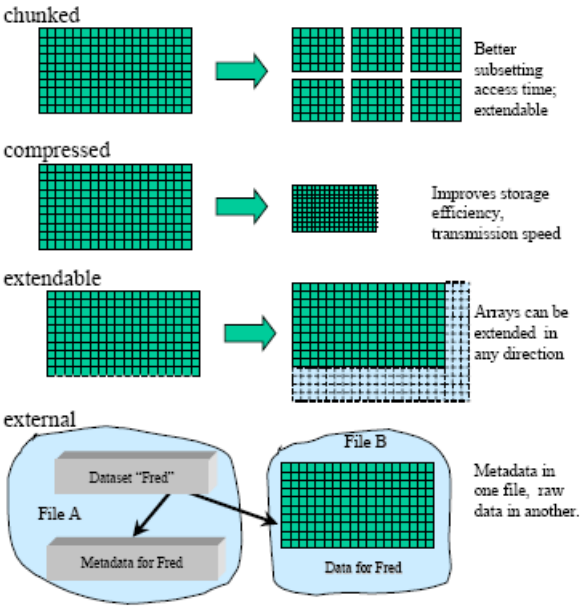


Figure 4.2: HDF5 Special Storage Options

The data can also undergo comprehensive transformations during the I/O process. These include datatype and location change, subsetting and optional separation of metadata from the raw data. The transformation operations could be performed either on a dataset or a datatype. In the first case a group of elements in a dataset is selected and modified. And in the second the datatype is changed. HDF5 supports a wide range of datatypes and also allows user-defined datatypes of any complexity and depth. HDF5 datatype is a set of properties stored within a data file. Figure 4.3 gives an insight into data spatial subsetting operations [Gro02].

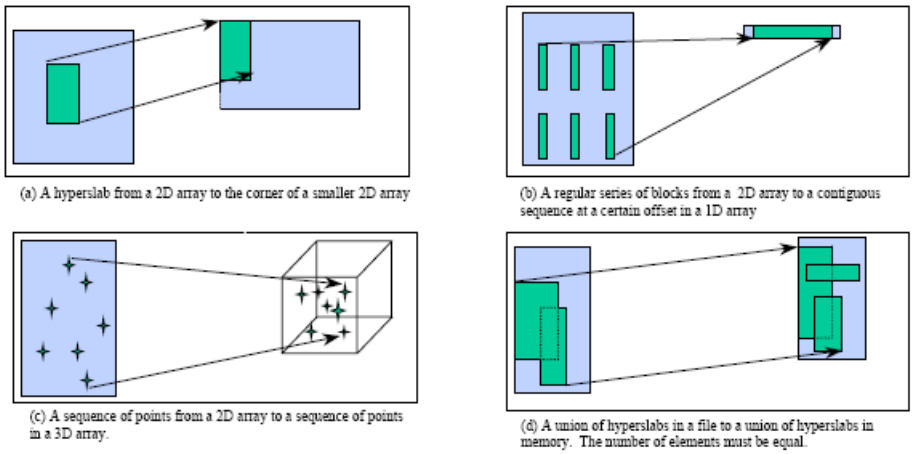


Figure 4.3: HDF5 Spatial Subsetting Operations

Although, HDF5 is a relatively new data format, it was warmly accepted in the scientific community and is widely used in various research fields. It is most suitable for the projects involving operation on large data quantities of diverse datatypes. This format makes a solid basis of the SAF, LibSheaf and CDMlib projects by the U.S. Department of Energy; SILO project at the Lawrence Livermore National Laboratory (LLNL); NASA EOS programme for studying the Earth's ozone, climate and air quality. It is used by the Swedish Meteorological and Hydrological Institute (SMHI), which has developed an HL-HDF5, a high-level interface for HDF5. And also at the Argonne National Laboratory for their Globus Project involving operations with computational grids.

4.2.3 Alternatives

The HDF5 main competitors are Portable Binary Data Format (PDB), Network Common Data Form (NetCDF), Flexible Image Transport System (FITS) and HDF4.

PDB from LLNL was designed for the U.S. Department of Energy and is widely used by physicists.

NetCDF was developed by Unidata Program Center with the intention to support the atmospheric science.

FITS started at NASA Goddard Space Flight Center as a project for processing of astronomical datasets.

HDF4 is the previous version of HDF. Similarly with HDF5 it was designed at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign (UIUC). Initially the format has been used to ease the centre's internal data exchange. And at later stages it was adopted by NASA for their Earth Observing System (EOS) programme.

Some of HDF5 properties are similar to those offered by alternative formats, but there are many spheres where HDF5 is more flexible and beneficial in comparison to its rivals. Table 4.1 summarises the main features of five data formats widely used in the scientific research. It is a reduced version of the format competitive matrix adopted from [Gro02]. It highlights the format features especially interesting for the current project.

4.3 Format Performance

The scientific format comparison would be incomplete without performance benchmarks. The experimental test results presented in [Gro02] complement the general picture. Figures 4.4(a) and 4.4(b) demonstrate performance of the sequential reading and writing operations with various buffer sizes. Each measurement includes timing of the following operations: opening the file and dataset, reading or writing the dataset, and closing the dataset and file.

Recall, that an HDF5 can use a split driver to store meta- and raw data independently. This approach allows a parallel computation based on HDF5 reaching almost the same level of performance as of native MPI I/O. Figure 4.5 depicts the benchmarks of parallel writing operations. Each process there writes a 10 MB large file.

Feature	HDF5	HDF4	NetCDF	PDB	FITS
Object Features:					
Files					
Parallel file access	Yes, on any system with MPI I/O	No	In development	Yes, on SMP machines	No
Storage media for the file	File system, memory, network and any other device using VFL	File system	File system	File system	File system
Scalability of objects					
Number of objects	Unlimited	Limited	Limited	Limited	Limited
Maximum size of objects/files	Limited only by computer or file system capacity	$2^{31}-1$ bytes	$2^{31}-1$ bytes	$2^{31}-1$ bytes	$2^{31}-1$ bytes
Object storage					
External storage of raw data	Yes	Yes	None	None	None
Arrays extendable in any direction	Yes	No	No	No	No
Metadata handling capabilities					
Ability to store metadata separately from raw data	Flexible	Limited	No	No	No
Support for complex metadata structures	Yes	No	No	Yes	No
Datatypes					
Simple integer and floating point datatypes	Integer and floating point types of any size or precision	Integer: 8, 16, 32, 64-bit signed and unsigned. Float: 32, 64-bit	Integer: 8, 16, 32-bit signed. Float: 32, 64-bit	Integer: 8, 16, 32, 64-bit signed. Float: 32, 64-bit	Integer: 8, 16, 32, 64-bit signed and unsigned. Float: 32, 64-bit
Compound datatypes (record structures)	Any amount of complexity, including types nested to any number of levels	One level, as part of the			
table structure	N/A	User-defined structures as in C language	One level, as part of the table structure		
Full datatype definition stored in file	Yes	No	No	Yes	No
User-defined datatypes	Yes	N/A	None	Yes	None
Library features:					
Storage in general (number of entries in a group)	Tunable B-tree ratios, group-to-object relationships	No	No	No	No
Subsetting of arrays	Hyperslabs, union of hyperslabs, point selections. Set operations on hyperslabs are designed in.	Simple hyperslab subsetting	Simple hyperslab subsetting	None	Simple hyperslab subsetting
Speed of access to objects	Fast	Slow	Moderate	Moderate	Slow
Data compression and other conversions	GZIP is built in, other compression methods or conversions could be added.	GZIP, JPEG, Adaptive Huffman compression	None	None	None

Table 4.1: Scientific Data Format Comparison

(a) Sequential Reading Benchmarks (b) Sequential Writing Benchmarks

Figure 4.4: Sequential Reading and Writing Benchmarks

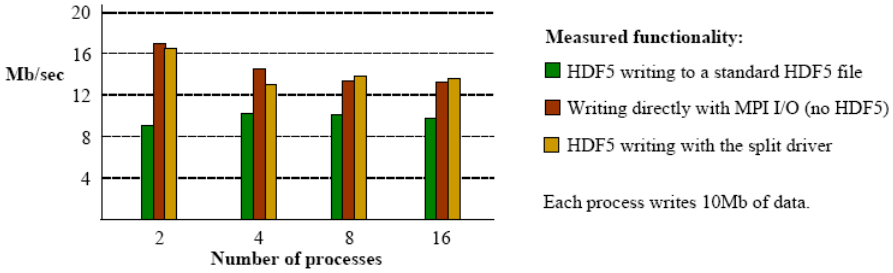
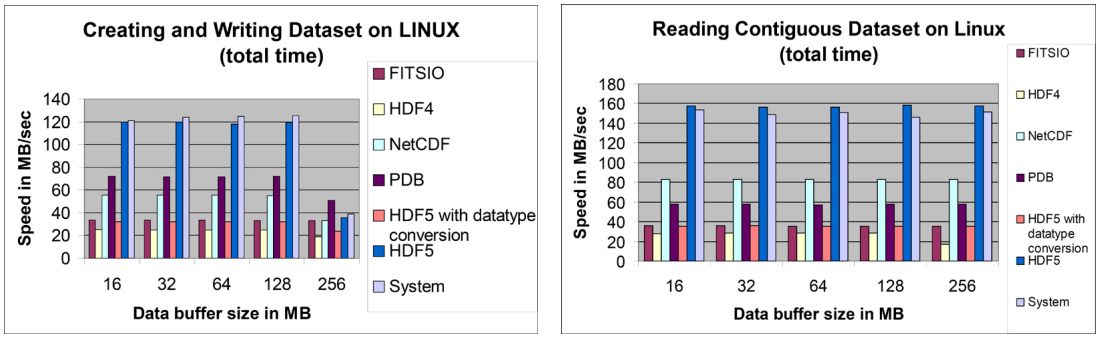


Figure 4.5: Parallel Writing Benchmarks

Another series of tests conducted by Elena Pourmal in [Pou02] shows a slightly different picture. The Figures 4.6(a) and 4.6(b) present performance measurements for creating, writing and reading a contiguous dataset on Linux. A single file of an appropriate format was created and then used to store a dataset. A dataset in that case was a two-dimensional array of unsigned short integers. The experiments were run for various dataset sizes from 2 up to 512 MB.

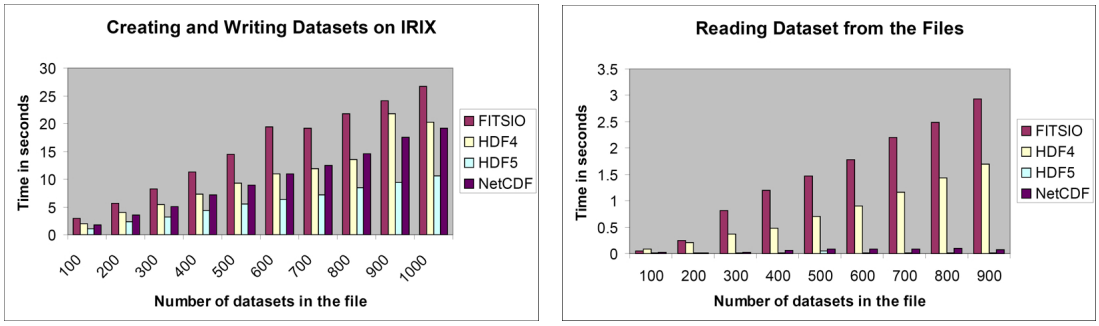


(a) Creating and Writing of Dataset (b) Reading of Contiguous Dataset

Figure 4.6: Contiguous Dataset Benchmarks

Operating upon a number of datasets is of special interest to this project because it reflects the data storage and access patterns of the current code. Figure 4.7(a) describes creating of multiple datasets and writing them into a file. Again, a

dataset in this experiment was represented by a two-dimensional array of unsigned short integers. A different number of equal datasets, each 1 MB large, was written into a file. The quantity of datasets written ranged from 100 to 1000. Finally, Figure 4.7(b) provides timing results on reading the last dataset from this file. Although, it might seem a somewhat artificial scenario, it gives a general impression of data lookup and access time.



(a) Creating and Writing of Multiple Datasets (b) Reading of Last Dataset in File

Figure 4.7: Multiple Dataset Benchmarks

4.4 Format Choice

Another option was an in-house binary format development. This idea would allow to build a highly customisable format aimed specifically for the FD-FDTD data storage and processing. But this flexibility comes at cost of time for defining the specification, design and development of the new format. The in-house implementation would not be a standard, would not have parallel support in first versions and most importantly its performance level would be unknown.

On the other hand the performance of modern scientific formats is thoroughly tested. It is proved that the data conversion time from ASCII to XDR and any other binary representation is minor in comparison to the disk access time (refer to Section 4.2.1). Moreover the conversion time becomes negligible when compared to the data production phase during the main FD-FDTD simulation or post-processing.

Building the simulation upon the HDF5 format requires a certain amount of time for reading the documentation and getting to know the library's API. But HDF5 is a reliable and tested platform. It is one of the widely accepted and used formats

for scientific data manipulation. Format's speed is comparable and often higher than of most of its competitors. At the same time the HDF5 file access time is slightly larger than of pure system (see Section 4.3). This format is not tied to Fortran only, but also supports C and Java interfaces. This gives flexibility for future developments and possibility for interconnection of different programs using the same data. HDF5 provides full support for parallelism and a number of command-line utilities for data viewing and transformation. Finally, there is a large user community and a dedicated help centre at the NCSA, which can help answering difficult questions. This format has bright prospecs for inclusion into contemporary database systems as one of the standard datatypes. HDF5 support is already offered by many high performance computing clusters in the world.

In general, HDF5 provides the highest degree of collaboration among all of its competitors. System-independent file format makes HDF5 datasets highly portable. These files could be accessed from machines with different memory and storage architectures. HDF5 is a perfect method for research data standardisation. Putting in place conventions on the way to organise and store data and metadata enables high data exchange, sharing and reuse. Having comparable or higher performance level with all of its rivals HDF5 excels in a much broader feature support. This format's unique properties like optimised parallel processing, unlimited file size, chunking, compressing and complex datatype support are critical in electromagnetic computation area. These capabilities of HDF5 grant enough freedom for finding the most effective solution for the current task and will serve a sound basis for the future work.

4.5 Summary

The evolution of data processing area will bring a smart notebook approach for daily data access and management. Current developments are focused on integrating ideas from file and database systems as well as programming languages into a single product. This will result in a number of benefits for end-users mainly reducing the overhead for data processing and enabling development of better data analysis and visualisation tools. Implementation of scientific data formats as default datatypes in modern databases will provide scientists with classical database features as indexing, data independence and parallel access.

Currently MPI and OpenMP offer a reliable alternative for parallel operations upon scientific data. There is a number of scientific data formats being developed concurrently. Some of them are of general and others of specific application character. The HDF5 data format was chosen for the purposes of this project. The format was selected because of its outstanding performance, flexibility in data definition and manipulation, and especially the ability for parallel data operations and Fortran programming interface.

Chapter 5

Development of Post-Processing Utilities

The route to efficient FD-FDTD data production and optimised post-processing lays in a number of subsequent modifications to the current in-house software. Since the ASCII format was considered inappropriate for keeping the large volumes of simulation data it was decided to replace it with the HDF5. After the design of an HDF5 output file's structure the main program should be altered to print out the data in a new form. While the final goal of this project is the development of the parallel data-postprocessing utilities, the sequential versions of both plot pointing and plane visualising algorithms are the necessary milestones in achieving this aim.

It is a well-known fact that parallelisation of an existing sequential algorithm could potentially hinder the development of an efficient parallel implementation. Therefore if possible it is recommended to design a parallel algorithm version from a scratch. That's why the sequential variants of the FD-FDTD post-processing methods are designed as a parallel implementation run on a single processor. Moreover, a thorough analysis of possible parallelisation approaches for the data plotting and animating utilities is made prior to any algorithm development.

Chapter 5 is one of the key parts of this thesis. This chapter deals with the implementation issues of this research work. The description of the simulation output file in terms of HDF5 begins in the Section 5.1. Section 5.2 contains the explanation of how the existent in-house software has been changed to accommo-

date the new output capability. The extensive analysis of possible parallelisation approaches for the data post-processing is presented in Section 5.3. The novel plot production and plane visualisation algorithms are described in Sections 5.4 and 5.5, which conclude the chapter.

5.1 Output File Structure in HDF5

Recall the general structure of the simulation's output data (see Section 3.2). The FD-FDTD program prints out the information in a group of files, whereas each file contains the electromagnetic values for the entire grid space, i.e. a file is a spatial representation of the electromagnetic signal. The FD-FDTD numerical calculation consists of a certain number of time steps. An output file is produced for each simulation time step. Then a collection of the simulation files will refer to a temporal signal representation. If the information kept in a single file describes a cubic grid space, the file itself could be compared with a *snapshot* of this cube for a particular time moment. The cube's form stays constant, but the electromagnetic wave moves within this cube. This wave propagation is reflected by a series of time snapshots depicting the signal status.

There is a column-wise organisation of data stored in a file (refer to Table 3.1). An output file could contain up to 12 data columns, where the first three are obligatory – they keep the x, y, z cartesian coordinates of a spatial position. The following 9 columns are optional – they are intended for storing the required displacement $\vec{D}_{x,y,z}$ and the electromagnetic field $\vec{E}_{x,y,z}$ and $\vec{H}_{x,y,z}$ values.

To design a new output file structure in HDF5, it is necessary to remember the four building blocks of the scientific format: group, dataset, datatype and dataspace (see Section 4.2.2 for more details). Since the spatial form of the simulation grid is defined by the cartesian coordinates, in three dimensions the FD-FDTD space could be viewed as a large cube, whereas at each grid point there is a data record keeping the simulated electromagnetic values.

In terms of HDF5 this structure could be defined as one global *dataset* storing the complete volume of information. This dataset will be described by a three-dimensional *dataspace*. A dataspace is comparable to a coordinate system. Since a number of electromagnetic field parameters should be stored for a single dataspace location, the *datatype* of an FD-FDTD dataset will be set to an array. Array

is not included into a set of default HDF5 datatypes. That's why it should be created as a custom user-defined datatype before the intended file use.

The library features as data chunking, shuffling and compression are applied to each HDF5 file created. Apart from allowing the efficient access to different portions of file data, chunking also enables the use of compression option. Dividing the dataset into chunks is vital for the independent parallel operations on the file data. All h5-files are compressed by means of the standard GNU zlib algorithm with the compression level set to 6. There exist 10 different levels of data encoding, where 0 defines no compression at all and 9 – the best possible archiving. The moderate compression level selected allows the relatively efficient data storage and at the same time it does not require large time for file encoding [CYCA03].

To provide an even better compression the HDF5 data is shuffled[1] before the zlib archiving. Since the locality of scientific data is relatively high, usage of shuffling algorithm will contribute to the higher compression ratio. However, the additional byte permutations for shuffling and re-shuffling of data during the compression and decompression operations might require more processing time.

The performance evaluation report conducted by the HDF5 Group [Gro04] states that on average 10% of compression ratio improvement is gained while using shuffling with gzip or bzip2 compression packages for the 32-bit float data and 5% for the 64-bit float data. This work also shows that less encoding and decoding time is required when applying shuffling and the bzip2 compression engine. It is also proved in the report that shuffling and bzip2 data compression needs less processing time than the sole bzip2 encoding of data.

5.2 Modification of Existing Software

Pseudocode in the Algorithm 5.1 highlights the major steps of the HDF5 output file production.

Variables used in the pseudocode notation are: i – specifying the HDF5 Fortran programming language interface, f_{h5}, g, s – denoting a new HDF5 file, the group

[1] Shuffling – is an approach for changing the byte order in a data stream, e.g. if the data consists of a set of numbers, when shuffled the first byte of each number will be stored in the first data chunk, second – in the second, etc. Shuffling is similar to the data interleaving widely used in computer networking.

1: **initialise** HDF5 Fortran interface i
2: **create** new file f_{h5}, group g, 3D dataspace s
3: **create** file array datatype t_{file}, memory array datatype t_{memory}
4: **modify** dataset creation property list l to use filters $r_{chunk}, r_{shuffle}, r_{zlib}$
5: **create** dataset $d \in g \in f_{h5} \leftarrow s, t_{file}, l$
6: **create** output and input data buffers $wdata, rdata$
7: **fill in** $wdata \leftarrow \vec{D}_{x,y,z},\ \vec{E}_{x,y,z},\ \vec{H}_{x,y,z}$
8: **write** $wdata \rightarrow f_{h5}$ using t_{memory}
9: **read** $f_{h5} \rightarrow rdata$ using t_{memory}
10: **close** l, s, d, g, f_{h5}, i

Algorithm 5.1: HDF5 Output File Production

created in this file and a three-dimensional dataspace. Parameters t_{file} and t_{memory} stand for custom user-defined array datatypes for file and memory access respectively. Variable l sets a dataset creation property list. Various data chunking, shuffling and compressing filters applied to the creation property list are denoted with parameters $r_{chunk}, r_{shuffle}$ and r_{zlib}. Letter d specifies the main dataset in the file f_{h5}, where the output data will be stored. Finally, variables $wdata, rdata$ contain the output and input data buffers, and the standard $\vec{D}_{x,y,z}$, $\vec{E}_{x,y,z}$ and $\vec{H}_{x,y,z}$ notation is used to specify the displacement and electromagnetic field values produced during the FD-FDTD simulation.

An HDF5 output file for the in-house software is created in the following way. First the Fortran interface is initialised to provide access to the HDF5 library subroutines. Then the new binary file in HDF5 f_{h5} is created. The group g takes up the second layer in the file group hierarchy after the default root group "/". This is done to allow more flexibility in manipulation of datasets in a single file. The three-dimensional dataspace s is created using the x, y and z output coordinate ranges of the simulation. Next two custom datatypes for the program data are created. These are the file and memory array datatypes t_{file} and t_{memory}. HDF5 intentionally distinguishes between these two datatypes. The special *file* datatype provides a platform-independent binary file representation. The latter *memory* datatype is different for each system, but it enables faster memory operations on data. Finally, the alteration of the default dataset creation property list l for the support of library chunking, shuffling and zlib compression options is done by adding the appropriate filters $r_{chunk}, r_{shuffle}, r_{zlib}$ to the list l.

All the above procedures comprise the preparatory phase of the HDF5 library usage. Now the main entity of an HDF5 file – the dataset d is created. This is a target storage place for the actual simulation data. The dataspace s, file datatype

t_{file} as well as the modified dataset creation property list l should be specified for a successfull creation of a dataset d. This dataset is located in the group g in a file f_{h5}. Then the data buffers $wdata, rdata$ are initialised. These are applied for a more efficient single access data I/O operations in the HDF5 file. In this case writing and reading of a dataset occurs in a one step manner, when the data is processed as a single chunk. After numerical part of the simulation is finished the $\vec{D}_{x,y,z}$, $\vec{E}_{x,y,z}$ and $\vec{H}_{x,y,z}$ values for the requested output grid space are stored into the $wdata$ variable. Then the contents of the intermediate output buffer $wdata$ are written into a particular file f_{h5}. The memory array datatype t_{memory} should be used during the data transfer. Reading of the filled in HDF5 file is optional. It is currently performed to validate the correct result of data writing operation. Access to all HDF5 variables should be terminated at the end of data output. This allows to release resources.

The existing in-house FD-FDTD software was modified to support the HDF5 file output. The HDF5 simulation data is written as described in the Algorithm 3.1. The original program capability to produce output in the ASCII format was left intact. This was done mainly for experimental and performance testing purposes. The ASCII data output will become optional or will be completely removed in the future revisions of the program.

Each major step of the HDF5 output file production is supplied with a proper status-message. This contributes for easier identification of errors and is especially useful during the code debugging.

5.3 Possible Parallelisation Approaches

The key to efficient implementation of data post-processing tasks consists in the idea of parallel execution. Therefore to enable simultaneous processing the FD-FDTD data should be distributed among available CPUs. For the resulting program efficiency it is vital that the workload division occurs in the right manner. The data chunks assigned to particular processors should be made independent of each other. This is required to avoid processing bottlenecks, where one CPU would have to wait for the result of another. This section provides an extensive analysis of possible workload distribution and parallelisation approaches. Understanding the data dependency character and the potential for multiprocessing

is crucial for the success of new plotting and visualising algorithms.

In general, there exist three workload distribution dimensions for the FD-FDTD data. The information could be separated among the participating processors according to (i) one of the grid space coordinate axis x, y, z, (ii) time step value t, and (iii) displacement $\vec{D}_{x,y,z}$ or electromagnetic field $\vec{E}_{x,y,z}$, $\vec{H}_{x,y,z}$ value. Considering the task of plot production, two cases of data post-processing might be distinguished. These are either plotting the *entire domain* or the *subdomain* of the simulation data. The following sections analyse both of the plot production cases.

5.3.1 Case I – Plotting Entire Data Domain

Plotting the entire data domain refers to producing plots of all displacement and electromagnetic field values for the entire grid space. In this case the most suitable workload distribution approach would be according to one of the coordinate axes. For example, if the data is divided among the z axis, where $z \in [1, 100]$, time step $t \in [1, 100]$ and processor rank $r \in [1, 4]$, then each processor is assigned 25 z-points. But the data post-processing in this way requires parallel file access. Since one output file represents the electromagnetic state of the entire grid space for a particular time moment, coordinates from all four z-ranges must be read from the same file. This is because each processor starts the preparation of the `plot.data` file from the very beginning of the simulation. This means all 4 CPUs will first need to access data files with time step $t = 1$, then $t = 2$, etc. Since the FD-FDTD data is stored in HDF5, this issue could be solved using the library subroutines for parallel file access.

Distributing the workload according to the time step value t will result in 4 processors responsible for the complete grid space each, where a specific CPU will obtain 25 successive time steps t. No parallel access to a single output file will be required in this case. But this work division will produce 4 separate `plot.data` files for each grid space point. This will happen because every processor will be capable of creating the plotting file only for its own time step range, i.e. r_1 will generate the `plot.data` files for all grid space coordinates in the time range $t \in [1, 25]$, $r_2 - t \in [26, 50]$, $r_3 - t \in [51, 75]$ and $r_4 - t \in [76, 100]$. Afterwards all 4 plotting files should be combined to obtain the final graph of the signal in the complete time domain $t \in [1, 100]$.

Spreading the post-processing load according to one of the displacement or electromagnetic field values makes little sense while it provides no uniformity. It will be complicated to distribute the variable number of $\vec{D}, \vec{E}$ or $\vec{H}$ values among the available processors evenly.

5.3.2 Case II – Plotting Data Subdomain

While plotting the entire data domain is a relatively rare task, it serves a good example for describing possible parallelisation strategies. On the contrary it is frequently needed to plot a single grid space point or a collection of specific points. By depicting the signal at certain key locations only can justify the correctness of the complete FD-FDTD simulation. Under assumption that it is required to plot the data subdomain, the most appropriate workload division will be according to the time step value t. This is true, while the detailed analysis of the other two parallelisation approaches reveals their unsuitability.

Distributing the target data among one of the grid space coordinates is complicated, while the number of points for plotting is hardly adjustable to the number of available processors, e.g. there might be only 2 grid space points required for plotting and 64 processors. The last possible workload division strategy is according to the electromagnetic field value. Again everything said for the previous case of coordinate axis distribution is true here. Assigning the variable number of $\vec{D}, \vec{E}$ and $\vec{H}$ output file columns to an unknown number of the post-processing CPUs is difficult. Reaching the evenness of processor load so important for an effective parallelisation is very hard using the coordinate axis or electromagnetic field values.

Review of the time step data distribution proves its flexibility. In this case all the points intended for plotting are assigned to every available processor. Since the data is spread according to the time step value t, no parallel access to a single output file will be required. The highest level of data independency is reached, while each CPU will operate on its own subset of output files. Using the time step dimension for data distribution makes each processor responsible for producing the plotting files in its own time step range. Although, this will result in a number of `plot.data` files for each target point of the grid space. A collection of plotting files for the same space point could be easily concatenated using the standard Unix utility `cat`.

5.4 New Plot Production

5.4.1 Design

The pseudocode in the Algorithm 5.2 explains the main phases of the new plot production implementation. Since plotting a single point or a small subset of grid space points is considered to be more practical, the algorithm was developed to distribute the workload according to the time step value t (see Section 5.3.2 for more details). The major improvements of this design over the initial algorithm are (i) the support of FD-FDTD output files in HDF5, (ii) capability to plot any amount of grid space points and (iii) the potential for efficient parallel processing of the simulation data.

1: **read** simulation parameters $T_g, n_{CPUs}, G_{co}, div_{axis} \leftarrow$ `params`
2: **read** coordinates of points to plot $C \leftarrow$ `points`
3: **determine** processor rank r for each point $p \in C$
4: **store** all distinctive CPU ranks $r \rightarrow R$
5: **generate** dat-file names $\rightarrow D$
6: **create** and open D
7: **initialise** HDF5 Fortran interface i
8: **for** each processor c responsible for points in C **do**
9: **for** each time step t in assigned time range T_l **do**
10: **generate** h5-file name n
11: **open** h5-file H_n, group g, dataspace s, datatype t_{memory}, dataset d
12: **read** complete dataset $d \rightarrow$ input buffer $rdata$ using t_{memory}
13: **for** each point $p \in C$ **do**
14: **if** (p was processed by c) **then**
15: **write** $t, p_e \rightarrow$ corresponding dat-file D_p
16: **end if**
17: **end for**
18: **close** d, t_{memory}, s, g, H_n
19: **end for**
20: **end for**
21: **close** D, i

Algorithm 5.2: New Plot Production

The variables used throughout the pseudocode are T_g, T_l for identification of *global* and *local* time step ranges. In this case global means the time step range for the entire FD-FDTD simulation and local stands for the range assigned to a post-processing CPU by an MPI master processor. Local time step range T_l is the main criterion for the workload distribution allowing effective parallelisation of

plot production. Parameter n_{CPUs} sets the number of available post-processing CPUs. The set G_{co} specifies the coordinate ranges in the workload division axis div_{axis} for each processor of the initial FD-FDTD simulation. Where the variable div_{axis} can take the values of x, y or z. All parameters T_g, n_{CPUs}, G_{co} and div_{axis} refer to the original FD-FDTD calculation and are read from an auxiliary simulation file `params`.

The set C keeps the coordinates of target grid space points intended for plotting. These point coordinates are passed by means of another file called `points`. Parameters r and p denote the single processor rank and target plotted point. Variable R stores a collection of all CPU ranks of the initial simulation. Parameters D and H identify the resulting plot data files and the set of HDF5 files, where D_p specifies a plot data file corresponding to a particular point p and H_n refers to a single h5-file with a name n. Variables i, c and t denote the HDF5 Fortran interface, single processor and time step values respectively. The HDF5 file parameters use the same notation as the pseudocode in the Algorithm 5.1. These are g for the file group, d for the dataset described by the three-dimensional dataspace s and containing the data of type t_{memory}. Input buffer $rdata$ is used to fetch the h5-file data for the post-processing operations. Finally, parameter e stands for a generic displacement or electromagnetic field value intended for plotting and p_e denotes an e value for a particular grid space point p.

The working manner of the novel algorithm is described in the subsequent paragraphs. First the initial FD-FDTD simulation's parameters are read from the auxiliary file $params$. The values $T_g, n_{CPUs}, G_{co}, div_{axis}$ will help to distribute the workload among the available post-processing resources. The idea behind these variables is the automatic identification of the original simulation's data production style. Knowing the initial way of data processing enables the efficient and correct load division for the post-processing activities. This is especially useful when the number of post-processing CPUs is not equal to the amount of simulating CPUs. No human interference is required during the organisation of the post-processing activity.

The next step is reading the space grid point positions intended for plotting. These are stored in the point coordinate set C. Afterwards the processor ranks R are determined for each point p kept in the set C. This is particularly important for the identification of a specific HDF5 file H_n, which will hold the electromagnetic field value e of a point p. Another preparatory step prior to the main

functional body of the algorithm is the plotting file name D generation and file opening. Since each processor is responsible for its own local time range T_l there will be n_{CPUs} plotting files for every grid space point p. The post-processing CPU rank is reflected in the plotting file name to distinguish between different dat-files and to be able to concatenate them in the right order.

After the HDF5 Fortran interface i is initialised the algorithm enters into the main phase of plot file production (lines 8-20, Alg. 5.2). There are two general loops in the code. Where the first allows to iterate through all initial processors responsible for the simulation of target points. The latter passes through all time steps t in the local time range T_l. Both of these cycles are required for the identification of correct HDF5 simulation files because the actual grid space points with the accompanying electromagnetic field data are stored in the specific h5-files, dependending on various parameters of an initial FD-FDTD simulation. Where the ongoing time step and the processor rank are crucial in identification of a particular HDF5 file.

The specific CPU rank and the time step value allow to construct the target HDF5 file name n. Then the file H_n and group g are opened. The dataset d is accessed after its dataspace s and the memory datatype t_{memory} are read from the group. Finally, the entire information domain contained in the dataset d is sent to the input buffer $rdata$. Then the current time step t and the electromagnetic field values e of each point p intended for plotting are forwarded to the corresponding plot data file D_p. Afterwards, when all the effective electromagnetic information has been extracted from the array $rdata$, the HDF5 dataset d, memory array datatype t_{memory}, dataspace s, group g and the file H_n are closed to release computing resources for the next h5-file. The ready plot files D and the HDF5 Fortran interface i are closed in the end of post-processing.

5.4.2 Implementation

The new plot pointing functionality described by the Algorithm 5.2 is implemented in Fortran 90 using the HDF5 library version 1.6.5. The program is called `plotPoints.F90`. A wrapper script `h5compile.sh` should be applied to the source code for the compilation. This is a custom bash script developed for convenient usage of HDF5, GNU zlib library and MPI Fortran compilers. The executable file requests no command-line arguments. All information required for

successful plot file production is stored in the text files `params` and `points`, whereas the first keeps the information from the initial FD-FDTD simulation and the latter specifies the total number of points to plot as well as their coordinates.

Table 5.1 depicts the contents of the simulation parameter file `params`. Values kept in the upper part of the table are especially important for the future post-processing. All other variables describe the physical settings used during the FD-FDTD calculation. The pseudocode notation of the Algorithm 5.2 is given in the left column for the easier recognition of parameter meaning. Table 5.1 gives an example for the FD-FDTD simulation performed for the time step range $T_g \in [1, 10]$, where two processors $n_{CPUs} = 2$ were used to produce the output data. The ranks of these processors as well as the coordinate ranges they were responsible for are stored below: $c_1 = 0$, $lr_1 \in [1, 5]$; $c_2 = 1$, $lr_2 \in [6, 10]$. The next row specifies the workload division axis $div_{axis} = 3$, which refers to z axis in this case. Then follow the simulation-wide output coordinate ranges: $r_x \in [1, 10]$, $r_y \in [1, 10]$ and $r_z \in [1, 10]$. In other words, there were two CPUs conducting the FD-FDTD calculation for the $10 \times 10 \times 10$ grid space for 10 time steps. The z axis was used for the data distribution, whereas each of the processors performed one half of the entire simulation: $lr_1 \in [1, 5]$, $lr_2 \in [6, 10]$.

Table 5.2 reflects the contents of the point coordinate file `points`. This file specifies the grid space coordinates of points intended for plotting. First table row sets the total number of plotted points. The rows below define the x, y, z coordinates of each point. In the example shown in Table 5.2 there are 3 points aimed for plotting: $p_1 = (1, 4, 3)$, $p_2 = (7, 2, 9)$ and $p_3 = (5, 5, 5)$.

The simulation data files in HDF5 should be accessable from the same directory where the program is started. Launching the new plotting program `plotPoints` will produce one data file per point per processor, i.e. for the example case reviewed in Tables 5.1 and 5.2 there will be 3 points $\times$ 2 processors = 6 plotting data files after the code execution. Also the data file naming convention was changed to better reflect the file contents. Previously, the `plotPoint.sh` script was capable of producing only one plotting file `plot.data`. This was inappropriate for the case of plotting many grid space points. The new file name is constructed according to the following template: `<x>-<y>-<z>_<rank>.dat`, where the `<x>`, `<y>`, `<z>` stand for the x, y, z point coordinate values and `<rank>` states the current rank of the post-processing CPU. This naming scheme allows for easy distinguishing of produced files. Table 5.3 shows the file names for 6 plotting files

produced in the example described in Tables 5.1 and 5.2.

Notation	Description	Values		
T_g	time step range	1	10	
n_{CPUs}	number of processors	2		
c_1, lr_1	rank, range	0	1	5
c_2, lr_2	rank, range	1	6	10
div_{axis}	division axis	3		
r_x	x range	1	10	
r_y	y range	1	10	
r_z	z range	1	10	
soft source case media parameters read from $param00000.bin$				
σ	sigma	0.2		
ε_{00}	epsilon00	2.8		
ε_s	epsilons	4.8		
τ	tau	0.7×10^{-11}		
	elements per wavelength	300		
	bytes per memory element	7488		
	wave frequency	0.3×10^{10}		
	grid size	10	10	10
	source location	5	5	5
	width	2.5		
	freqmodu	6.0		
	isourcecase	2		
	impulse location	5	5	5
	time	0	0.4×10^{-02}	

Table 5.1: Structure of Simulation Parameter File $params$

Some parallels might be drawn between the pseudocode parameters in the Algorithm 5.2 and the variables implementing them in the program. The set G_{co} storing the coordinate ranges along the division axis of the original FD-FDTD simulation is realised by means of a two-dimensional array variable `ranges()`, where the dimension sizes are $n_{CPUs} \times 3$. This results in a three-column data structure, while the first column keeps the processor's rank c and the remaining two store the starting and ending coordinate range values. The pseudocode parameter C reflecting the plotted point coordinate values refers to another two-dimensional array `co()` of size $n_{points} \times (d_{rank} + 1)$. Here the d_{rank} specifies the dataset's rank in the HDF5 output file, which is 3 for a three-dimensional dataset. One record in the data structure `co()` will store the initial processor's rank c responsible for the current point production and the point location in

3		
1	4	3
7	2	9
5	5	5

Table 5.2: Structure of Point Coordinate File $points$

1-4-3_00000.dat
1-4-3_00001.dat
7-2-9_00000.dat
7-2-9_00001.dat
5-5-5_00000.dat
5-5-5_00001.dat

Table 5.3: Plot File Names

x, y, z grid space coordinates. The variable R, which keeps the distinctive ranks of processors participated in the original simulation, is implemented with help of a one-dimensional array `ranks()` of length n_{points}. Due to the inability to create variable size arrays in Fortran 90, this data structure contains n_{points} entries [Smi95]. Each record in `ranks()` stores either the CPU rank or value "-1", indicating the empty field. The set of the resulting plot files is denoted by parameter D (Alg. 5.2), which is realised with another compound data structure of length n_{points} called `plot_file()`. The program variables `ranges()`, `co()` and `ranks()` are of type integer, while `plot_file()` contains character data entries of length 64. These code variables are very important for the automatic mapping of original simulation workload distribution to the number of processors available for the plot production.

Also a number of additional subroutines were developed to lighten the plot file creation and manipulation tasks. The methods `get_valency()`, `to_str()` and `to_zstr()` were implemented for the convenient file name generation, where `get_valency()` returns the quantity of digits in a number, but `to_str()` and `to_zstr()` convert integer values into character string or a zero-lead string. A mathematical function was designed for the `get_valency()` procedure:

$$f(x) = \lfloor \lg(x) \rfloor + 1,$$

where $f(x)$ denotes the quantity of digits in a number x. Finally, the subroutine `exists()` verifies whether a given element exists in a set.

5.4.3 Drawbacks

One of the potential drawbacks of the current point plotting implementation is that the entire dataset is read from the HDF5 file into the input buffer $rdata$. This could reduce the program performance, especially in the most frequent execution cases, when only a small amount of space points have to be plotted. The overhead of reading the complete dataset compared to reading only the certain points could be relatively large. Although, the HDF5 library offers wide means for partial data selection such as separate points and hyperslabs, implementing directed reading of specific points was complicated and error-prone. This happened partly because the HDF5 library and APIs are written in C programming language and the internal HDF5 data is stored in row-wise approach, which is native to C. Since the developed plotting software was implemented in Fortran, which arranges the data in a column-wise order, it was very difficult to read the C-native HDF5 data specifying the Fortran-native dataspace indices [EPL94]. In order to provide a fully-functional plotting utility it was decided to use the less efficient, but correct way of reading the HDF5 data.

Another performance issue is the HDF5 speed of search for a dataset element. It is unknown how the dataset is parsed for a single element. Currently there is no indexing feature support for HDF5 datasets. Creating an index for each dataspace dimension x, y, z would significantly reduce the element look-up time. Indexing is one of the new improvements scheduled for the next HDF5 release 1.8.0. It is awaited that this software version will be available to public in autumn 2007.

5.4.4 Jumping Approach

An improved element look-up algorithm was developed in terms of this project. It was called the *jumping approach*. The idea of this searching technique lays in the subsequent advancement through the dataspace dimensions. Since the spatial position of a target point is known prior to the electromagnetic field data extraction it could be used for an intelligent look-up of a required value. At first the FD-FDTD simulation coordinate ranges should be analysed and the respec-

tive axes should be sorted in the descending order. This enables traversing the grid space from the largest to the smallest coordinate range. Finding the target point's coordinate value in the largest range will perform a "jump" to a certain section of the HDF5 dataset. Since the other coordinate ranges are smaller it is assumed that the target point's electromagnetic value will be relatively close to the current dataset location. Afterwards another "jump" should be made to find the point's coordinate value in the second largest axis. This will further reduce the point look-up scope. Finally the last "jump" performed in the vicinity of first two coordinate values will obtain the required field value.

Table 5.4 gives an example of the simulation output file contents. The target point's coordinates will be $p_{x,y,z} = (117, 356, 92)$. In this case the global coordinate ranges are $x \in [1, 189];\ y \in [1, 467];\ z \in [90, 95]$. Sorting the axes in descending order according to the range size will result in the following sequence: y, x, z. Now applying the jumping approach will help to find the point's electromagnetic values. First, the algorithm will "jump" to the position 356 in the y axis, afterwards it will look for the coordinate 117 in the x axis and finally "jump" to the value of 92 in the z axis. This will place the HDF5 internal file pointer on to the target point's record $117, 356, 92$. The last processign step is acquiring the electromagnetic field values.

Because of the project time constraints this approach has not been implemented practically.

Space Coordinates			Field Values		
x	y	z	$\vec{E}_x$	$\cdots$	$\vec{H}_z$
1	1	90	-0.15396E+10	$\cdots$	-0.15041E+12
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
117	356	92	0.11483E+9	$\cdots$	0.64315E+11
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
189	467	95	0.13878E+08	$\cdots$	0.13895E+10

Table 5.4: Output Data File Structure

5.4.5 Parallelisation

Due to the limited time amount allocated for the project only the sequential version of the new plotting utility was produced. However, this sequential implementation was specifically developed for future parallelisation. The program parallelisation should be reached by even distribution of post-processing workload between the participating CPUs. The load division should occur according to the global time step range T_g kept in the simulation parameters file `params`. The efficient code parallelisation could be achieved in two ways: (i) by making the MPI master processor responsible for the initial workload distribution or (ii) keeping each available CPU independent and forcing it acquiring all the necessary data on its own. In the first case the MPI master processor will read the `params` and `points` auxiliary files and spread the load among the other CPUs by sending them their specific local time ranges T_g as well as the target point coordinates x, y, z [VAN99]. On the contrary in the second case the `params` and `points` data files should be also stored in HDF5. This will enable parallel access to the subsidiary files for each participating processor. Which one of these two approaches is faster is a question of future implementation and performance experiments. The first case is better from the implementation complexity point of view.

5.5 New Visualisation Production

5.5.1 Improvement of Sequential Version

This section presents two variants on the way to improve and speed up the existent sequential script `visualisePlane.csh`. Both improvements are depicted by means of pseudocode. The mathematical notation of the pseudocode is consistent throughout the dissertation and most of the symbols pertain their meaning used in the previous Algorithms 3.1, 3.2 and 3.3. The pseudocode in Algorithm 5.3 shows the first series of optimisations. The main difference between the proposed code and the original in the Algorithm 3.3 is contained in new operations in lines 5-7 and 8-13. Instead of 6 times using the `sort` utility for finding minimum and maximum values of each coordinate axis and the electromagnetic fields the new version applies `gawk` to parse the entire output file once. During

this single file tracing all the required maximum and minimum values are collected. However, the performance improvement of this case should be verified by experiments. It is still possible that the `sort` utility implements a very efficient sorting algorithm, which applied 6 times might be faster than a single file parsing with `gawk`.

1: **pass** coordinate value v_{a_3}, processor rank r, data files F to `gawk`
2: **for** each file $f_r \in F$ **do**
3: **use** `gawk` to find plane l_{a_1,a_2} specified by v_{a_3}
4: **output** $e \rightarrow$ `file.pgm`
5: **for** each file column $c \in (c_{a_1}, c_{a_2}, c_e)$ **do**
6: **set** $v_{max,c}, v_{min,c} \leftarrow r_{0,c}$
7: **end for**
8: **for** each file row w **do**
9: **for** each file column $c \in (c_{a_1}, c_{a_2}, c_e)$ **do**
10: **use** `gawk` to find $v_{max,c}, v_{min,c}$:
11: **if** $(v_c > v_{max,c})$ **then** $v_c \rightarrow v_{max,c}$
12: **else if** $(v_c < v_{min,c})$ **then** $v_c \rightarrow v_{min,c}$
13: **end if**
14: **calculate** $(v_{max,c} - v_{min,c} + 1) \rightarrow cg_c$
15: **if** $(cg_c > g_c)$ **then**
16: **replace** g_c with cg_c in `file.pgm`
17: **end if**
18: **end for**
19: **end for**
20: **use** Fortran program to convert `file.pgm` $\rightarrow$ `file.ppm`
21: **end for**
22: **use** ImageMagick to convert `*.ppm` $\rightarrow$ `movie.mpeg`

Algorithm 5.3: Improved Visualisation Production, Variant A

Another option to make the initial visualisation algorithm performing faster is shown by pseudocode in the Algorithm 5.4. This program modification could be applied to the visualisation problem under assumption that the simulation grid space size is constant for every time moment. This is true for the current FD-FDTD method implementation. Lines 2-7 and 11-16 (Alg. 5.4) depict the main changes in the visualisation code. In this case the 4 program calls to the sorting utility could be performed only once for the entire post-processing task. Running the `sort` command for a single time on the first output file f_{r_0} will obtain the necessary maximum and minimum values v_{max}, v_{min} for each grid space axis a_1, a_2. Since the grid dimensions remain constant throughout the simulation, the first 4 calls to the `sort` utility could be excluded from the main program cycle (lines 8-17). There will be only 2 sorting calls left. These are still needed for finding the

electromagnetic field value range cg_{c_e}, while these values are different for each output file. Using 2 `sort` calls instead of 6 for each simulation file should drastically reduce the overall post-processing time. Similarly, if the grid space sizes do not change throughout the FD-FDTD calculation, the minimum and maximum axis values could also be read from the `params` file, which keeps all the general information about the simulation. This would further increase the visualisation performance, saving time on another 2 calls to the Unix sorting utility at the beginning of the post-processing activity.

1: **pass** coordinate value v_{a_3}, processor rank r, data files F to `gawk`
2: **use** file f_{r_0} :
3: **for** each file column $c \in (c_{a_1}, c_{a_2})$ **do**
4: **use** `sort` to find $v_{max,c}, v_{min,c}$
5: **calculate** $(v_{max,c} - v_{min,c} + 1) \rightarrow g_c$
6: **store** $g_c \rightarrow$ `file.pgm`
7: **end for**
8: **for** each file $f_r \in F$ **do**
9: **use** `gawk` to find plane l_{a_1,a_2} specified by v_{a_3}
10: **output** $e \rightarrow$ `file.pgm`
11: **use** `sort` to find v_{max,c_e}, v_{min,c_e}
12: **calculate** $(v_{max,c_e} - v_{min,c_e} + 1) \rightarrow cg_{c_e}$
13: **if** $(cg_{c_e} > g_{c_e})$ **then**
14: **replace** g_{c_e} with cg_{c_e} in `file.pgm`
15: **end if**
16: **use** Fortran program to convert `file.pgm` $\rightarrow$ `file.ppm`
17: **end for**
18: **use** ImageMagick to convert `*.ppm` $\rightarrow$ `movie.mpeg`

Algorithm 5.4: Improved Visualisation Production, Variant B

5.5.2 Parallelisation

The pseudocode in the Algorithm 5.5 highlights the major features of the parallelised plane visualisation functionality. All the previous theoretical as well as practical ideas are put together in the design of this new parallel algorithm. Similarly to the plot production task the simultaneous multiprocessing is achieved through worklad division according to the time step value. Again this way of parallelisation was selected because it is more frequently required to visualise only a small number of simulation planes. Usage of HDF5 attributes, which will "stamp" each simulation data file with the maximum and minimum values in every grid size dimension will allow to elude any data sorting and parameter

file access prior to the main post-processing. The coordinate extremum values will be read just before their intended usage (line 11, Alg. 5.5). In this case only the displacement or the electromagnetic field values $\vec{E}$ should be acquired from the HDF5 dataset d. The library hyperslab selection feature will be applied to obtain the specific portion of the effective data. While there is no indexing capability implemented in the current version of the HDF5 software, the maximum and minimum value look-up in the input buffer $rdata$ should be developed manually (line 13). The `max()` and `min()` subroutines could be either implemented at the HDF5 library data access level or as an ordinary Fortran 90 procedure. The extremum value searching functionality could be improved at the later development stages, when the HDF5 indexing feature becomes available to the general public. It is strongly believed that maximum and minimum value look-up will be performed faster when relying on a valid data index.

Finally, usage of Netpbm[2] command-line utilities `pgmtoppm` and `ppmtompeg` will contribute to another reduction of post-processing time. Since Netpbm is a native specification package for PGM and PPM file formats it is awaited that its tools provide the most efficient media conversion commands. Netpbm data transition utilities should replace the custom Fortran 90 program as well as the ImageMagick generic `convert` command. Moreover Netpbm is one of the software packages that comprise the standard major Unix/Linux distributions.

5.6 Summary

The current in-house software developed in Fortran was enhanced with the HDF5 subroutines to support data output in this scientific format. Each simulation output file is built upon the three-dimensional dataspace using the user-defined array datatype for storing the displacement and electromagnetic field values. Chunking, shuffling and compression filters were applied to the dataset creation property list for further improvement of data storage efficiency. Due to limited time availability only the sequential version of the point plotting utility was implemented. However, the sequential implementation was designed with the necessary future parallelisation in mind.

[2]Netpbm – is a toolkit created for effective manipulation of graphical images. Comprised of about 300 command-line-based utilities it is especially aimed for convenient image format conversion. Current version supports over 100 different image formats.

1: **read** simulation parameters $T_g, n_{CPUs}, G_{co}, div_{axis} \leftarrow$ `params`
2: **read** locations of planes to visualise $L \leftarrow$ `planes`
3: **determine** processor rank r for each plane $l \in L$
4: **store** all distinctive CPU ranks $r \rightarrow R$
5: **for** each processor c responsible for planes in L **do**
6: **for** each time step t in assigned time range T_l **do**
7: **generate** h5-file name n
8: **open** h5-file H_n, group g, dataspace s, datatype t_{memory}, dataset d
9: **use** t_{memory} :
10: **read** $\vec{E}$ from dataset $d \rightarrow rdata$
11: **read** $v_{max,a_1}, v_{min,a_1}, v_{max,a_2}, v_{min,a_2}$ from dataset attribute $u \rightarrow attr$
12: **for** each plane $l \in L$ **do**
13: **use** `max()` and `min()` on $rdata \rightarrow e_{max}, e_{min}$
14: **calculate** ranges g_{a_1}, g_{a_2}, g_e
15: **write** $l_e, g_{a_1}, g_{a_2}, g_e \rightarrow$ corresponding pgm-file P_l
16: **end for**
17: **close** d, t_{memory}, s, g, H_n
18: **use** `pgmtoppm` to convert $P_l \rightarrow$ ppm-file Q_l
19: **end for**
20: **end for**
21: **use** `ppmtompeg` to convert $Q \rightarrow$ visualisation V_l

Algorithm 5.5: New Visualisation Production

Different parallelisation approaches that could be potentially employed for simultaneous multiprocessing were analysed. Two most suitable workload distribution strategies were discovered. It was shown that dividing the load according to the coordinate axis value is the preferred option for the case of entire data domain processing. On the contrary the load should be spread according to the time step value range while plotting or visualising the data subdomain.

The novel implementation of the point plotting functionality supports the HDF5 simulation files, is capable of plotting any number of the grid space points and comprises a strong basis for future parallelisation.

There were two major improvements proposed for the current plane visualisation algorithm. One of them relies on the single parsing of the simulation file with `gawk` programming language instead of using Unix sorting utility for 6 times. While another proposal lays in moving 4 program calls to the `sort` command out of the main loop.

Finally the parallel visualisation algorithm was developed. The key features of the new plane visualisation are the application of HDF5 attributes for storing the grid space extrema, development of new `max()` and `min()` program functions and extensive usage of Netpbm utilities for the data format conversion and final visualisation production.

Chapter 6

Experiments on High Performance Computing Systems

This chapter concentrates on the large-scale computational electromagnetics which simulates the wave propagation in the vicinity of the Vivaldi antenna array elements. First the architectures of the supercomputer Bull system at the University of Manchester (called Horace) and the IBM BlueGene/L at the University of Groningen in the Netherlands are presented. After discussing the current experience with these systems and highlighting their drawbacks a thorough analysis of the potential performance of these machines is made.

6.1 Horace

6.1.1 Architecture

Horace is a high performance computing system based in the Manchester Computing facility at the University of Manchester. It consists of 192 processor cores based on the Bull Itanium 2 architecture. The overall amount of processor cores is spread between 24 compute nodes. Each of these nodes includes 4 Itanium 2 Montecito Dual Core 1.6 GHz processors with 8 MB of cache per processor. This means every Horace compute node offers 8 cores for computation. There are 16 GB of RAM and 512 GB of scratch disk space available for each node. There exist 4 special purpose nodes with high memory configuration of 32 GB of RAM.

All 24 compute nodes are connected by a single rail Quadrics QsNetll (elan4) interconnect. The whole system has a 10 TB large total filestore. Figure 6.1 represents the general architecture of a single computing node in Horace.

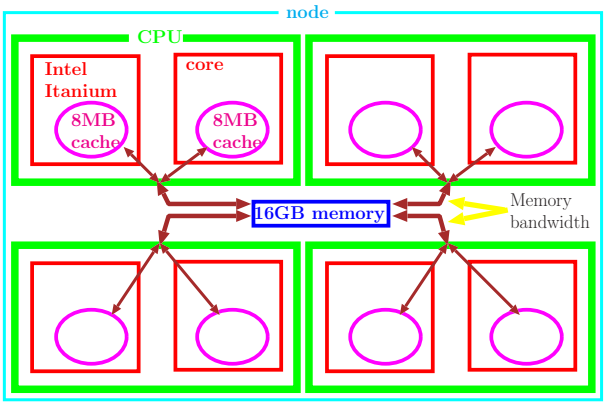


Figure 6.1: Computing Node Architecture in Horace

6.1.2 Current Experience

Horace offers solid computing resources backed-up by a sufficient amount of operating memory. Currently this system runs most of the research group's FD-FDTD simulations. But the scientific computation demand in the Manchester Computing is very high. This puts serious workload on the system and consequently restricts the amount of computing time available per individual user. The job submission queues are long. Most of group's experiments are run using only 2 normal specification computing nodes resulting in 16 processing cores and 32 GB of operating memory. Requesting this amount of computing power results in up to 3 days of waiting time for a single job.

6.1.3 Drawbacks

As expected with supercomputers the low-throughput is the issue. To increase the frequency of user access to Horace resources the 24 hour job limit was put in place. Each user is provided with 1 GB of personal disk space. Both the job running time limit and the amount of user space on Horace are tight and bind the FD-FDTD simulation potential. However, it is possible to use the temporal 3 TB scratch disk area or obtain access to reserve storage space under special consideration. For the time being the group's simulation data is copied from

Horace to local cluster machines. The next drawback of the Horace architecture is a weak I/O mechanism.

6.1.4 Potential Performance

Two series of experiments were conducted on Horace to estimate the system's potential performance. The first series included the simulation of a single Vivaldi antenna element. This simulation was conducted for 5000 timesteps and 16681707 grid points, since one antenna element occupies $189 \times 467 \times 189 = 16681707$ space grid points. All 6 electromagnetic field values $\vec{E}_x, \vec{E}_y, \vec{E}_z, \vec{H}_x, \vec{H}_y$ and $\vec{H}_z$ for the complete grid space were output into files. These real-world experiments were run on 2 computing nodes (16 processor cores) with 32 GB of operating memory. Refer to the section called "Single Vivaldi Antenna Element" below for further details about this series.

The second series was comprised of 6 experiments for various cubic space grid dimension sizes: 150, 250, 350, 450, 550 and 650 points. All experiments in this series were conducted for 100 simulation timesteps. Only $\vec{E}_z$ field values for the whole grid space were output into files. The same hardware resources on Horace were used to perform the second series of experiments – 2 computing nodes and 32 GB of RAM. Detailed analysis and discussion of this series could be found in the section named "Scientific Probing".

Single Vivaldi Antenna Element Running the simulation for a single Vivaldi antenna element allowed to estimate the potential operating memory usage. According to the Horace command line utility `bjobs -l` the amount of RAM used for this task was 1.768 GB. This means the amount of memory required to store one space grid point for the simulation is 1.768 GB $/$ 16681707 points = 1.06×10^{-7} GB. Knowing the amount of memory required to store a single grid point and the overall amount of RAM available to each computing node allows drawing further assumptions. The potential maximum dimension size for a cubic space grid that could be efficiently calculated on the Horace system equals to $\sqrt[3]{\frac{32\text{ GB}}{1.06\times10^{-7}\text{ GB}}} \approx 670$ points. Another interesting result is a prediction of the overall amount of antenna elements that could be calculated on Horace without using the disk swap area. This equals to $670^3/16681707 \approx 18$ antenna elements.

The simulation produces one file per timestep per processor. Currently the output is produced in ASCII form and consists of 9 field records. The fields of a single file record are the x, y and z coordinates, and $\vec{H}_x, \vec{H}_y, \vec{H}_z, \vec{E}_x, \vec{E}_y$ and $\vec{E}_z$ electromagnetic field values. Each file produced was approximately 200 MB large. The amount of disk space required to accommodate the total simulation data would be 5000 timesteps $\times$ 8 processors $\times$ 200 MB $= 7.6294$ TB. And this is not yet the maximum amount of information that could be stored in an output file.

It took approximately 6 minutes to output one file for a time step on each processor. The complete calculation of 5000 timesteps would have required 5000 files $\times$ 6 minutes $= 500$ hours ≈ 21 days.

Scientific Probing Figure 6.2 presents the total memory requirements for the simulation. The theoretical memory usage was estimated based on the data obtained from the simulation of a single Vivaldi antenna mentioned earlier. A series of experiments for the grid dimension sizes of 150, 250, 350, 450, 550 and 650 resulted in the experimental curve on the Figure 6.2. It is important to notice that theoretical assumptions represent the worst-case memory usage scenario. The experiments show that real memory demands are less intensive and it would be possible to simulate a cube with the grid dimension larger than 670 points and hence accommodate more than 18 antenna elements. The intersection of the theoretical and experimental curves is the value obtained from the single Vivaldi antenna simulation.

Figure 6.3 depicts the single output file size according to the grid dimension.

Refer to Figure 6.4 to observe the total processing time of a single timestep according to grid dimension size. The processing time of a timestep is the summation of the field calculation and output production times for a particular timestep. The processing time was measured for a single CPU.

The processing time of 35 minutes for 550 and 58 minutes for 650 grid dimension sizes is high in comparison to 1 and 3 minutes for 150 and 250 dimension sizes. This significant increase of processing time might be caused by the weak I/O mechanism of Horace, which might not be suitable for the I/O intensive computations. Also similar to the IBM BlueGene/L (see 6.2.3) there might be less I/O nodes than the computing nodes in Horace. At some point the I/O nodes might become highly overloaded. However this problem needs further investigation. A series of experiments should be conducted with and without output of electromagnetic values.

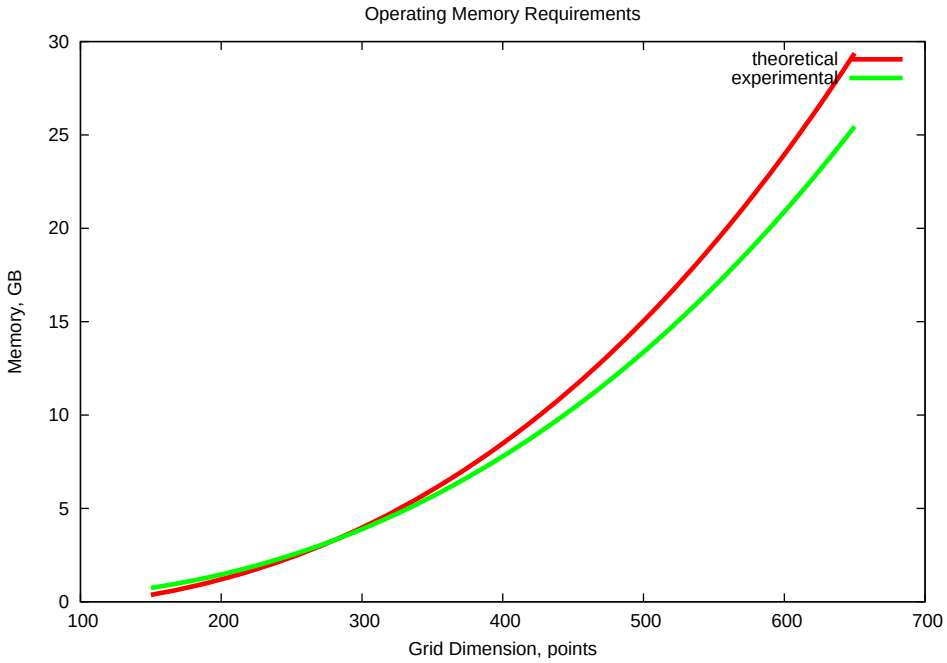


Figure 6.2: Operating Memory Requirements

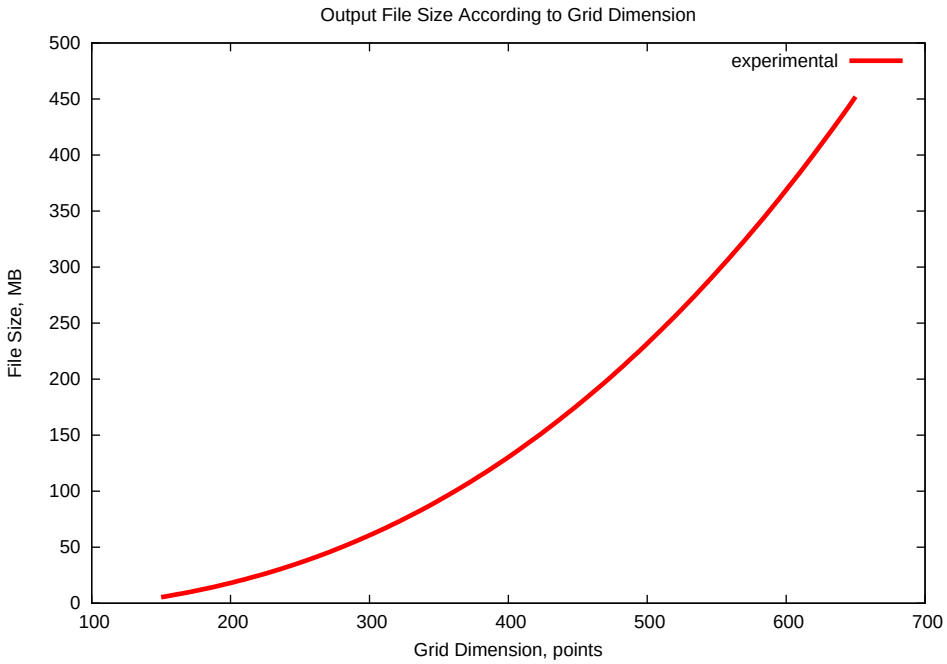


Figure 6.3: Output File Size According to Grid Dimension

Figure 6.5 highlights the total time requirements for a complete simulation. Depending on the grid dimension size and hence the single timestep processing time the overall simulation time increases with the number of FDTD space grid points. The theoretical curve's values were calculated from the experiment with a 150 dimension size. The experimental curve does not follow the theoretical one because of the 24 hours job running time limit in Horace. The experiments were successfully finished for 150, 250 and 350 grid dimension sizes. The 450, 550 and 650 were terminated by the Horace scheduling system, because they required more than 24 hours of simulation time.

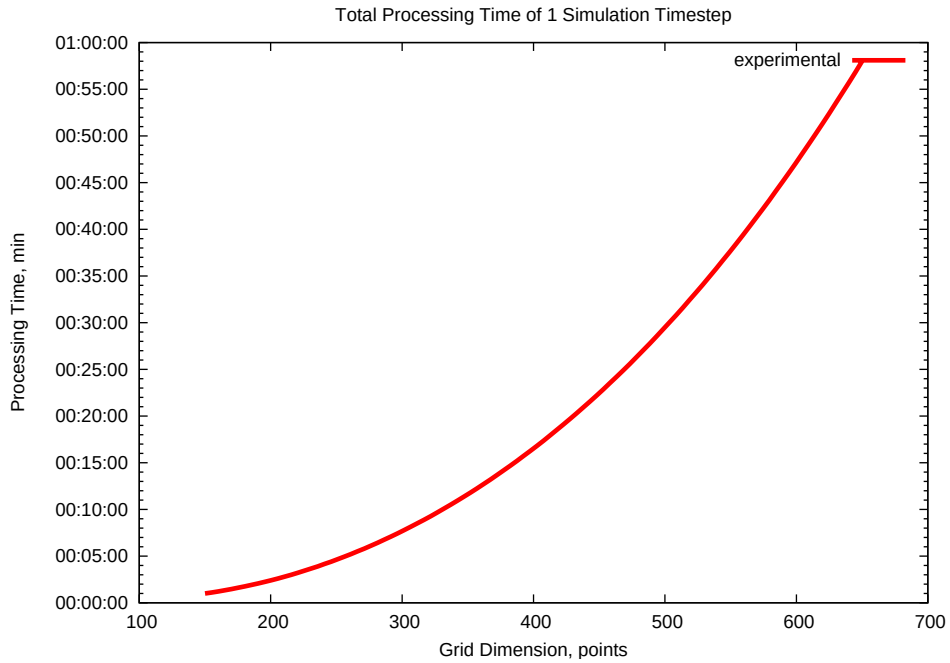


Figure 6.4: Total Processing Time of 1 Simulation Timestep

6.2 IBM BlueGene/L

6.2.1 Architecture

IBM BlueGene/L (BGL) is a massively parallel supercomputer architecture developed by IBM in collaboration with Lawrence Livermore National Laboratory (LLNL). The particular system intended for usage in current Frequency-Dependent Finite Difference Time Domain (FD-FDTD) simulation is the IBM BlueGene/L im-

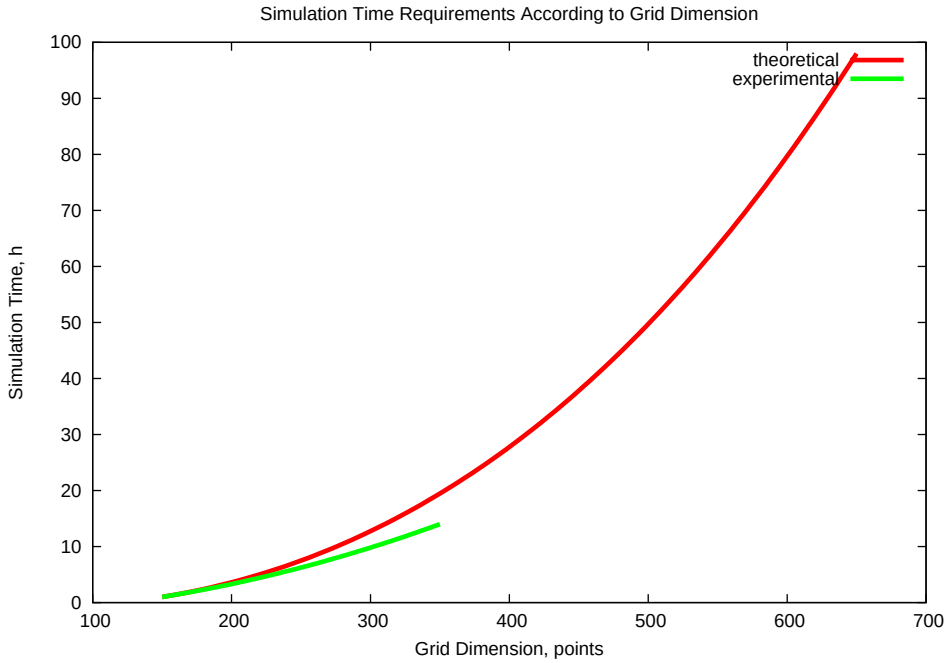


Figure 6.5: Simulation Time Requirements According to Grid Dimension

plementation for the Astron/Lofar project. This machine is based on the premises of the University of Groningen in the Netherlands. The Astron/Lofar system consists of 6 racks, where each rack includes 128 I/O nodes and 1024 compute nodes.

Every compute node is comprised by the 32-bit dual core 700 MHz Power PC 440. It has a customised Floating Point Unit (FPU) called "Double Hammer" and 512 MB of RAM. The node runs a proprietary single-threaded micro operating system.

The I/O machines have identical hardware configuration to the compute nodes. The only difference is that these systems are operated by the Linux OS with a communication daemon. One I/O machine is responsible for the I/O operations of 8 compute nodes. Aimed for high speed and large volume I/O operations these systems are also equipped with 1 Gb ethernet connection. Figure 6.6 represents the overall IBM BlueGene/L architecture. The compute card on this scheme is a single compute node described previously.

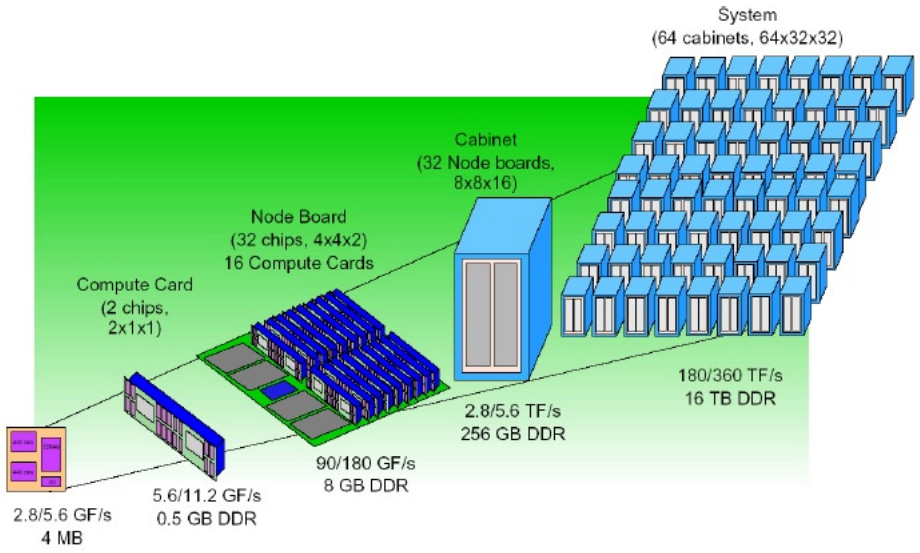


Figure 6.6: IBM BlueGene/L System Overview

6.2.2 Current Experience

First of all the original FD-FDTD code has been tailored to make it work on the Astron/Lofar machine in the Netherlands. The main simulation code is written in Fortran and uses C and MPI subroutines. It is compiled with the Intel Fortran complier both on the local research group's cluster and Horace supercomputer system. The IBM BlueGene/L provides special cross-compilers for its system. Each cross-compiler's name is preceded with the abbreviation XL, e.g. XL Fortran compiler. There is a number of compile scripts which wrap each compiler and provide it with library paths, libraries and compilation flags [GMSS07]. This makes the situation more complicated. There are `mpixlf95`, `xlf95` and `blrts_f95` Fortran compile scripts. The `Makefile` of the in-house software was changed to use the `mpixlf95` which refers to parallel XL Fortran 95 compiler [Lan04].

The program compilation procedure on the IBM BlueGene/L system is different. There are two types of nodes on BGL: *front-end* and *back-end*. The front-end machines are used for code's compilation. Each user is assigned to one of eight front-end machines. After successful compilation the executable file should be copied to the back-end machine and submitted into the job queue by means of special command-line utilities. Detailed information on BGL and Horace job manipulation commands is given in Appendix C. This separation into front- and back-end nodes is intended for keeping the back-end nodes solely for computa-

tion tasks, avoiding possible interference in resource usage and disk access operations between users. Designed for better system use and workload distribution this feature makes the program compilation and software installation procedures more complicated. This is because the front- and back-end machines can have very different hardware. The main task of the cross-compiler is to tie the code compiled on the front-end node with the hardware resources available on the back-end machine. The Astron/Lofar system had no HDF5 installed previously. The cross-compiler system and restricted user access made it impossible to install the HDF5 library on the BGL. Numerous telephone and e-mail communication sessions with the BGL system administrators and HDF5 developers have not brought positive results yet.

Previously the simulation grid space size and the number of processors were passed to the Intel Fortran compiler by means of the `-D` compiler option, e.g. `-D np=4`, which would set the number of processors for the simulation to 4. The IBM compilers have other options making passing the program arguments more difficult. This issue was solved by setting the runtime parameters directly in the main source code. However alternative solutions exist. These parameters could be read from an input file before the start of the simulation.

Some modifications were made to support the C subroutine calls in the the Fortran code on BGL. First the original file extention ".F90" was changed to ".F". Also the subroutine names in both Fortran and C source codes were made identical and changed to comply with the following naming convention:

```
<subroutine_name>_
```

Afterwards the program was successfully compiled using the parallel XL Fortran 95 compiler.

It is possible to launch computing jobs in two different modes on IBM BlueGene/L: *co-processor* and *virtual node* mode. By default the jobs are run in the co-processor mode. In this case the two cores of a CPU are treated as a single processor. This results in faster MPI communication between processors and larger amount of RAM available per CPU – 512 MB. In contrary, the virtual node mode *sees* two cores of a processor as two independent CPUs. This will offer twice as much processing power for a computing job, but also imply slower MPI communication and most importantly reduce the amount of operating memory per processor. Since the FD-FDTD electromagnetic simulations require both large

amounts of RAM and processing power, all research group experiments were run in the default co-processor mode.

Simple test simulations without HDF5 support were run on the BGL. Unfortunately, due to the mis-configuration of the resource management system, only up to 32 compute nodes were used. Configuration errors are currently being fixed.

6.2.3 Drawbacks

The Astron/Lofar BGL system has some known drawbacks. First of all, there are 8 times less I/O nodes than computing nodes, since one I/O node is responsible for input and output of 8 computing nodes. Hence I/O intensive computation can easily damage the whole system.

Secondly, each computing node has only 0.5 GB of memory. Since the current code developed for Horace was designed to work with 2 GB of memory per node, it is necessary to modify the program to run successfully on 512 MB of RAM on the IBM BlueGene/L in Astron.

Despite of these problems as well as minor configuration errors of the resource management system, the BGL offers an enormous amount of computing power. This makes it a unique high performance system with invaluable potential for the FD-FDTD simulations.

6.2.4 Potential Performance

Based on the experimental data obtained from Horace (see 6.1.4) it is possible to estimate the maximum number of antenna elements and cubic grid space dimension that could be efficiently simulated on the Astron/Lofar machine. The maximum amount of processors that could be requested on the BGL for one job is 1024. Each node on the IBM BlueGene/L has 512 MB of operating memory. Therefore, the total amount of RAM available for simulation will be 512 GB. Now dividing the total RAM by the RAM required for simulation of one antenna element will give the potential maximum of antenna elements that could be calculated on the BGL system: $512\ \text{GB} \,/\, 1.768\ \text{GB} \approx 289$ antennas. The maximum size of the cubic space grid equals to $\sqrt[3]{\frac{512\ \text{GB}}{1.06\times10^{-7}\ \text{GB}}} \approx 1690$ points.

6.3 Summary

The world of high performance computing resides on the three whales: (i) available amount of CPUs and operating memory, (ii) efficiency of the I/O mechanism and (iii) the quality of the load balancing subsystem. The overall performance picture is complicated by the interdependencies which exist between the I/O efficiency and the amount of RAM and also the I/O efficiency and the load balancing subsystem.

Finite Difference Time Domain (FDTD) is very CPU and RAM demanding algorithm. It puts high emphasis on the I/O mechanism. The solution to this is a trade-off between the simulation's performance, speed and precision. Some of these components have to be compromised in order to produce reasonable simulation results.

Chapter 7

Future Work

Chapter 7 discusses the project areas where a further work could be done to improve the currently achieved status. Section 7.1 suggests the potential actions to undertake for reaching these improvements. While Section 7.2 summarises the main research stages and concludes the work.

7.1 Project Improvement

Extensive research work has been undertaken in the context of the project. However only the sequential version of the point plotting utility was practically implemented. The most important goal of the future work is the fully functional parallelised variants of both point plotting and plane visualisation programs. Two approaches exist for parallelisation of point plotting tool: (i) with an MPI master processor distributing the workload and (ii) with independent CPUs accessing the parameter files and identifying their part of the work. Both of these options should be implemented and the most efficient selected for the final software version.

A large amount of practical experience with HDF5 library and High Performance Computing systems was gathered during the research work. The parallel post-processing utilities could be built upon the successfully developed plotting code. Separating the workload and running the post-processing procedures on many processors will greatly reduce the time consumption of data-processing. More time and effort should be invested in installing the HDF5 library on the IBM

BlueGene/L system in the Netherlands. Another series of code optimisation and adjustment for this particular supercomputer might be required.

The novel post-processing tools should undergo a thorough testing procedure, where the new sequential and parallel versions should be compared with the existing software. The timing results of the FD-FDTD simulation data output in ASCII and in HDF5 should be gathered and analysed. Different compression engines and levels should be applied to the HDF5 output to identify the most suitable in terms of the I/O speed and the storage space requirements. One of possible future research topics is the reduction of the I/O workload. The in-house software could be enhanced with additional measurement subroutines tracking the time spent for each major step of the simulation. Another innovation could be an intelligent output approach. Similarly to the ideas proposed by Jürgen Schnack for the approximate Lanczos diagonalisation [SHS07] the FD-FDTD program could print out only the existing electromagnetic field values. This means the time will be saved on all zero field values.

There are many aspects of the work that require further investigation such as the study on whether or not the program output time depends on the file size and format. This could be done by performing simulations with the gradual increase of output, e.g. simulation without any output, with one field value only E_z, and with six field values $\vec{H}_x, \vec{H}_y, \vec{H}_z, \vec{E}_x, \vec{E}_y$ and $\vec{E}_z$.

The impact of HDF5 usage on overall storage space demand is another issue which needs experimental support. It is obvious that application of HDF5 will be beneficial, but the degree and character of this benefit should be tested in greater detail.

The inefficient reading of the entire HDF5 dataset should be replaced with reading of the required grid space points only. Therefore further experiments on HDF5 library subroutines are needed. It is essential to find a correct way of acquiring the C-native data using the Fortran interface.

Also some slight improvements could be made to the in-house software itself. First of all dividing the program into Fortran subroutines and modules will increase the transparency, usability and re-use of the FD-FDTD code. The amount of the simulation input files should be reduced to the necessary minimum. Finally, the original ASCII and the new HDF5 output capabilities should be implemented as Fortran subroutines.

7.2 Conclusion

This project addressed the performance issues of FD-FDTD data post-processing. This research work had a dual goal of improving the data plotting and visualisation efficiency, and testing the potential computation and simulation performance of two supercomputer systems: Horace and BGL. The experiments conducted on High Performance Computing systems have shown the large potential of these machines for the electromagnetic simulations as well as the data post-processing activities. It was estimated that Horace could potentially calculate up to 670 point large cubic grid space which refers to 18 Vivaldi antenna elements. The BGL is capable of simulating the 1690 point large grid space, whereas it will accommodate 289 antennas. Despite of the difficulties in installing the HDF5 software on the BGL hardware and drawbacks in the job scheduling system, this machine provides a solid platform for the future FD-FDTD simulations.

Three phases were identified in the efficiency improvement of data post-processing activities. These were the modification of data output and storage procedure, development and implementation of novel sequential and parallel utilities for data plotting and visualisation purposes. Detailed analysis of the current in-house software revealed the slow speed and large disk space demands of the simulation output in ASCII format. After a thorough comparison of available scientific data formats it was decided to apply the HDF5 software for efficient storage and manipulation of the simulation data. HDF5 was chosen because of its flexibility, wide spectrum of unique data handling capabilities, Fortran programming language support and an optimised interface for parallel data access.

A detailed review of possible parallelisation approaches was conducted before the design of novel post-processing algorithms. The workload division according to the time step range was selected as the most appropriate for multiprocessor plotting and visualisation activities. The sequential version of the new plotting utility was developed in Fortran 90 using the HDF5 library APIs. The proposed point plotting software could be efficiently parallelised by introducing the MPI subroutines for the initial workload division. A series of performance drawbacks were found in the current implementation of the plane visualisation algorithm. Possible variants to improve the sequential version of the visualisation program were made. Finally a parallel visualisation algorithm was designed, but not implemented due to the time constraints of the research project.

It is possible to state that the novel approach of using the HDF5 data format and exploiting parallelism of data post-processing will bring a great improvement in speed of plot and visualisation production. This work also shares the practice of HDF5 implementation and usage as well as identifies possible performance ranges achievable on supercomputer systems in the domain of electromagnetic calculation.

Appendix A
Program Files

The list below contains a full register of the FD-FDTD simulation, data processing and complementary program files that could be found on the supplied CD-ROM. Each file name is followed by a short description of its intended purpose.

`mpi_Mur1.f` – original FD-FDTD simulation program implementing parallel calculation and ASCII output

`fdtd-mpi.F90` – modified FD-FDTD simulation program implementing parallel calculation and ASCII and HDF5 output

`plotPoint.sh` – original bash script for point plotting

`plotPoints.F90` – new plotting tool capable of processing HDF5 output files

`visualisePlane.csh` – original C shell script for plane visualisation

`bitTable.c` – C program to convert the media parameters from ASCII to the binary data format

`bitVector.c` – C program to convert the media parameters from ASCII to the binary data format

`bitVector.h` – C header file required for successful work of `bitTable.c` and `bitVector.c` code

`toolBox.h` – C header file required for successful work of `bitTable.c` and `bitVector.c` code

`Makefile` – text file used by the `make` utility for the compilation of the FD-FDTD code. It describes the relationships among program files and states the commands for updating each file.

`h5compile.sh` – bash script for the convenient compilation of Fortran programs using HDF5 library and MPI subroutines

`concat.sh` – shell script for concatenation of simulation output files produced by individual machines

`job.run` – bash script for the convenient launch of the parallel FD-FDTD simulation program

`proc2cpu` – auxiliary text file mapping available computing resources to the MPI processors used in the FD-FDTD simulation

`initialdata` – input file setting the FD-FDTD grid space size and spatial resolution

`width-freqmodu` – input file setting the pulse width and level of modulation

`impulsenumber` – input file setting the shape of a pulse

`impulselocation` – input file setting the location of the FD-FDTD grid space excitation, the interior space

`waveformnumber` – input file setting the shape of a pulse

`param00000.bin` – input file setting media parameters: static and optical permittivity, conductivity and relaxation time for each FDTD point

`points` – input file for the novel plotting utility `plotPoints.F90` setting the number and locations of plotted points

`data/h5/*.h5` – collection of FD-FDTD simulation output files in HDF5 format

`data/out/*.out` – collection of FD-FDTD simulation output files in ASCII format

`data/dat/*.dat` – collection of new point plotting tool `plotPoints.F90` output files in ASCII format

`data/cmd/*.cmd` – collection of Gnuplot command files for plotting the `plotPoints.F90` output files

`fort.25` – FD-FDTD code output file for the verification of the hard-source excitation waveform

`fort.26` – FD-FDTD code output file for the verification of the soft-source excitation waveform

`params` – auxiliary output file of the FD-FDTD code summarising the programming and Physical simulation parameters

Appendix B
Software Installation

Appendix B explains the installation procedure of SZip 2.1 and HDF5 1.6.6 software packages. All actions are performed for a user called "scofield". The target installation directory is `/home/scofield/software/64/`.

SZip

Download[1] the tarball source code of the latest SZip version and unpack it into the target directory:

```
tar -xzvf szip-2.1.tar.gz
```

Change to the newly created directory `szip-2.1`, configure, build and test the software code:

```
cd szip-2.1
./configure --prefix=/home/scofield/software/64/szip-2.1/
make
make check
make install
```

The SZip libraries will be installed into:

```
/home/scofield/software/64/szip-2.1/libs/
```

[1] SZip could be obtained from the HDF Group FTP server at `ftp://ftp.hdfgroup.org/lib-external/szip/2.1/`. Last accessed on September 6, 2007.

Add this directory to the environment variable `LD_LIBRARY_PATH` for convenient usage:

```
export  LD_LIBRARY_PATH=$LD_LIBRARY_PATH: \
        /home/scofield/software/64/szip-2.1/libs/
```

This command might also be saved into the user's bash shell parameter file:

```
~/.bashrc
```

HDF5

Download[2] the tarball source code of the latest HDF5 version and unpack it into the target directory:

```
tar -xzvf hdf5-1.6.6.tar.gz
```

Change to the newly created directory `hdf5-1.6.6`, configure and build the software code:

```
cd hdf5-1.6.6
CC=             "~/software/64/mpich2-1.0.5p3/bin/mpicc"
./configure     --prefix=~/software/64/hdf5-1.6.6/ \
                --enable-parallel \
                --enable-fortran \
                --enable-static \
                --disable-stream-vdf \
                --disable-shared \
                --with-szlib=~/software/64/szip-2.1/libs/ \
                --with-zlib=/lib64/:/usr/lib64/
make
```

[2]HDF5 could be obtained from the HDF Group FTP server at `ftp://ftp.hdfgroup.org/HDF5/current/src/`. Last accessed on September 6, 2007.

Start the MPI daemon on at least two machines to achieve the real parallelism. MPI service is needed for running a variety of HDF5 library tests to verify the installation correctness. Launch MPI daemon and obtain the service's host name and port number:

```
mpd &
mpdtrace -l  →  <host>-<port>
```

Login to a different machine and start the MPI daemon specifying the host name and port number of the first system:

```
mpd -h <host> -p <port> &
```

Verify and finalise the HDF5 software installation:

```
make check
make install
```

The HDF5 libraries will be installed into:

```
/home/scofield/software/64/hdf5-1.6.6/libs/
```

Add this directory to the environment variable `LD_LIBRARY_PATH` for convenient usage:

```
export  LD_LIBRARY_PATH=$LD_LIBRARY_PATH: \
        /home/scofield/software/64/hdf5-1.6.6/libs/
```

This command might also be saved into the user's bash shell parameter file:

```
~/.bashrc
```

Appendix C
Job Management Commands

Appendix C summarises the job management commands used on Horace and IBM BlueGene/L High Performance Computing systems for program manipulation. The user called "scofield" will execute all example commands shown in the following sections for a Fortran 90 program named `fox.f`.

Horace

The user has to log in before he will be able to submit a job to the Horace scheduling system:

```
ssh -X scofield@horace.manchester.ac.uk
```

Then a program file has to be transferred to the user's home directory on the system:

```
scp ./fox.f scofield@horace.manchester.ac.uk:~/fox/
```

Next the user's code is compiled with an MPI Fortran 90 compiler using the Unix `make` utility and a custom `Makefile`:

```
make fox
```

Finally the job is submitted into one of the available execution queues. This is done by applying the option `run` of the `make` utility. Whereas the exact action sequence is described by the `Makefile`:

```
make run
```

The Horace job submission command `bsub` is called when launching the program with the `make` utility. The command-line arguments specifying the job execution character are given in the list below:

`-W <mins>`	– wall time, maximum running time for the job.
`-c <cpu_time>`	– CPU time, maximum processor time for the job.
`-P <project>`	– name of the project to which the job belongs.
`-J <job_name>`	– assign a job name.
`-n <procs>`	– number of requested processors.
`-q <queue_name>`	– submit a job to a specific queue.
`-o <out_file>`	– write program output to a specified file.
`-e <err_file>`	– write error messages to a specified file.
`-w <dependency>`	– start another job after successfull finish of the first one.
`-R span [hosts=1]`	– force the job to execute on a single node.
`-B <email_address>`	– send an e-mail notification, when the job starts.

There is another complementary command `prun`, which is referred to by the `bsub` utility when launching an MPI program:

```
prun -n <nodes> <executable_file>
```

When the `make` utility starts the job, the `bsub` command is executed, which respectively calls the `prun` utility. The following line contained in the `Makefile` will submit a job to a default queue on Horace:

```
bsub -n 64 -W 60 -P river -o fox.out -J fox prun -n 8 fox
```

The program will request 64 processors on 8 nodes. The maximum running time will be 60 minutes. This job will belong to a project ”river“ and the output will be redirected to a file `fox.out`. The job will run under a name ”fox“.

Other important job manipulation utilities for the Horace system are given below:

`bsub`	– submits a job into a queue and assigns it a unique identification number.
`bjobs`	– returns job identifications and statuses of submitted jobs.
`-r`	– provides the information on running jobs only.
`-p`	– provides the information on pending jobs and gives the reasons for delays.
`-s`	– provides the information on suspended jobs.
`-l <job_id>`	– provides detailed information about a specific job.
`bpeek`	– returns an update of a job progress by showing the status of the standard output.
`-f`	– constant job monitoring for further updates.
`bkill <job_id>`	– terminates a submitted job.

IBM BlueGene/L

There are 8 front-end nodes on the BGL system: `bglfen0..7`. Each user is assigned to a specific node. System user "scofield" has two home directories, one on the front- and another on the back-end node:

```
/home/scofield/
/bglst1/home/scofield
```

First the user has to log in to his particular front-end node:

```
ssh scofield@bglfen1.service.rug.nl
```

Next the user compiles his program with the IBM XL Fortran 95 compiler `mpixlf95` using the Unix `make` utility and a custom `Makefile`:

```
make fox
```

Afterwards the produced executable file `fox` should be copied to the back-end:

```
cp fox /bglst1/home/scofield/river/
```

At this time the job could be submitted into an appropriate queue. Executing a command `partlist` identifies idle queues ready to accept user jobs. There are 4 types of queues. Each of them has a specific purpose determined by a maximum number of compute nodes and a running time limit. The queue descriptions are given in the table below:

Queue	Compute Nodes	Runtime, Hours
`default`	32	4
`long`	1024	24
`q72`	1024	72
`vlong`	1024	240

A command-line utility called `cqsub` should be used to submit a job into selected queue on the BGL. There is a number of parameters that should be specified while placing the job into the queue. The list below gives an overview of the required command options:

`-q <name>`	– queue name, values: `default`, `long`, `q72`, `vlong`.
`-C <work_dir>`	– working directory.
`-t <max_runtime>`	– maximum job running time in minutes, job will be terminated afterwards.
`-n <max_nodes>`	– number of requested computing nodes to run the job.
`-m <mode>`	– compute mode, values: `co` – co-processor, `vn` – virtual node.
`<path_to_prog>`	– full path to the executable file including the file name.

When the desired queue is identified the user can submit his job to the BGL scheduling system:

```
cqsub  -q long \
       -C /bglst1/home/scofield/river \
       -t 60 \
       -n 64 \
       /bglst1/home/scofield/river/fox
```

Launching the command in the specified way will submit the job into the long queue. The program directory will be `/bglst1/home/scofield/river`. The maximum running time for this job will equal to 60 minutes and the program `fox` will be executed on 64 computing nodes.

The table below gives additional information on other frequently used command-line utilities on the BGL:

`partlist`	– prints out the job queue list with the status of each queue.
`cqsub`	– submits a job into a particular queue, returns the job identification.
`cqstat`	– prints out the status of a job in a particular queue.
`cqdel <job_id>`	– terminates a job.

Appendix D
Source Code

Appendix D contains the source code listings for data output part of the modified FD-FDTD simulation program, new plotting utility, original plane visualisation script and the compilation script for parallel HDF5 programs.

Modified FD-FDTD Simulation Program *fdtd-mpi.F90*, Data Output Part

```
! BEGIN DATA OUTPUT

#ifndef no_output

      CALL MPI_BARRIER(MPI_COMM_WORLD, IERR)

      if (output_z_end .GE. output_z_init .AND.
         (t .EQ. 0 .OR. mod(t,output_modulus) .EQ. 0)) then

         zoff = output_z_offset
         zmin = output_z_init + zoff
         zmax = output_z_end + zoff

         filename = to_zstr(rank) // "_" // to_zstr(t) // ".out"
         print *, "file name: ", filename

         dirfile = trim(dirname) // trim(filename)

! Initialise the wdata array.

     print *, "zoff: ", zoff
```

```
      print *, "xmin: ", xmin, "ymin: ", ymin, "zmin: ", zmin
      print *, "xmax: ", xmax, "ymax: ", ymax, "zmax: ", zmax
      print *, "nf: ", nf

      allocate( wdata(xmax-xmin+1, ymax-ymin+1, zmax-zmin+1, nf), &
                stat=err )
      if (err .NE. 0) then
        print *, "wdata array allocation: failed"
        stop
      else
        print *, "wdata array allocation: succeeded"
      end if

! Output simulation data in ASCII format
! reason -> for HDF5 correctness check

 12   format(3I4, 1G15.5)
          open(unit=70, &
               file=dirfile, &
               access="sequential", &
               form="formatted", &
               status="new",iostat=ios)

          if (ios .NE. 0) then
            print *, "Error opening", dirfile, "for writting"
            stop
          endif

!     Initialise output buffer 'wdata'
      do i = 1, xmax-xmin+1
        do j = 1, ymax-ymin+1
          do k = 1, zmax-zmin+1
            do f = 1, nf
              wdata(i,j,k,f) = 0.0
            end do
          end do
        end do
      end do

!     Start conventional data  output and buffer fill in
      print *, "Entered main output loop"
      do i = xmin, xmax
        do j = ymin, ymax
          do k = zmin, zmax
```

```
          write(70,12) &
                i, j, k, &
                real( hx(i,j,k) )
        end do
      end do
    end do

    close(unit=70)

    do i = xmin, xmax
      do j = ymin, ymax
        do k = zmin, zmax
          wdata(i,j,k,1) = i
          wdata(i,j,k,2) = j
          wdata(i,j,k,3) = k
          wdata(i,j,k,4) = REAL( hx(i,j,k) )
        end do
      end do
    end do

    print *, "output buffer filled in successfully"

!   Initialize FORTRAN interface.
    CALL h5open_f(err)
       if (err .EQ. -1) then
         print *, "h5 interface initialisation: failed"
       else
         print *, "h5 interface initialisation: succeeded"
       end if

!   Create a new file using default properties.
    filename(12:)=".h5"
    dirfile = trim(dirname) // filename

    CALL h5fcreate_f(dirfile, H5F_ACC_TRUNC_F, file_id, err)
       if (err .EQ. -1) then
         print *, "h5 file creation: failed"
       else
         print *, "h5 file creation: succeeded"
       end if

!   Create a new group "/MyGroup"
    CALL h5gcreate_f(file_id, groupname, group_id, err)
       if (err .EQ. -1) then
         print *, "h5 group creation: failed"
```

```
      else
        print *, "h5 group creation: succeeded"
      end if

!     Create a dataspace.
      dims = (/xmax-xmin+1, ymax-ymin+1, zmax-zmin+1/)

      print *, "dataspace dims: ", dims

      CALL h5screate_simple_f(drank, dims, dspace_id, err)
      if (err .EQ. -1) then
        print *, "h5 dataspace creation: failed"
      else
        print *, "h5 dataspace creation: succeeded"
      end if

!     Create an array datatype to store EM-values.
      CALL h5tarray_create_f(H5T_IEEE_F32LE, dtype_rank, dtype_dim, &
                             dtype_id, err)
      if (err .EQ. -1) then
        print *, "h5 array datatype-f creation: failed"
      else
        print *, "h5 array datatype-f creation: succeeded"
      end if

      CALL h5tarray_create_f(H5T_NATIVE_REAL, dtype_rank, dtype_dim, &
                             dtypemem_id, err)
      if (err .EQ. -1) then
        print *, "h5 array datatype-m creation: failed"
      else
        print *, "h5 array datatype-m creation: succeeded"
      end if

!     Set dataset creation property list
!     to create chunked dataset and use zlib compression
      CALL h5pcreate_f(H5P_DATASET_CREATE_F, dset_prop_id, err)
      if (err .EQ. -1) then
        print *, "h5 property list creation: failed"
      else
        print *, "h5 property list creation: succeeded"
      end if

      xchunk = 5
      ychunk = 5
      zchunk = 5
```

```
      chunk_dims = (/xchunk, ychunk, zchunk/)

      CALL h5pset_chunk_f(dset_prop_id, chunk_rank, chunk_dims, err)
         if (err .EQ. -1) then
           print *, "h5 set chunking: failed"
         else
           print *, "h5 set chunking: succeeded"
         end if

      CALL h5pset_shuffle_f(dset_prop_id, err)
         if (err .EQ. -1) then
           print *, "h5 set shuffling: failed"
         else
           print *, "h5 set shuffling: succeeded"
         end if

      CALL h5pset_deflate_f(dset_prop_id, 6, err)
         if (err .EQ. -1) then
           print *, "h5 set zlib compression: failed"
         else
           print *, "h5 set zlib compression: succeeded"
         end if

!     Create a dataset with a customised property creation list.
      CALL h5dcreate_f(group_id, dsetname, dtype_id, &
                       dspace_id, dset_id, err, dset_prop_id)
         if (err .EQ. -1) then
           print *, "h5 dataset creation: failed"
         else
           print *, "h5 dataset creation: succeeded"
         end if

      print *, "file writing dims: ", dims

!     Write the dataset from an array wdata into file.
      CALL h5dwrite_f(dset_id, dtypemem_id, wdata, dims, err)
         if (err .EQ. -1) then
           print *, "h5 writing dataset into file: failed"
         else
           print *, "h5 writing dataset into file: succeeded"
         end if

         deallocate(wdata, stat=err)
         if (err .NE. 0) then
           print *, "wdata array deallocation: failed"
```

```
      stop
    else
      print *, "wdata array deallocation: succeeded"
    end if

    allocate( rdata(xmax-xmin+1, ymax-ymin+1, zmax-zmin+1, nf), &
              stat=err )
    if (err .NE. 0) then
      print *, "wdata array allocation: failed"
      stop
    else
      print *, "wdata array allocation: succeeded"
    end if

!   Read the dataset from a file into an array wdata.
    CALL h5dread_f(dset_id, dtypemem_id, rdata, dims, err)
    if (err .EQ. -1) then
      print *, "h5 writing dataset into file: failed"
    else
      print *, "h5 writing dataset into file: succeeded"
    end if

    do i = xmin, xmax
      do j = ymin, ymax
        do k = zmin, zmax
          print "( A, 3I2, A, 4G15.5)", &
                "rdata(", i, j, k, "):", rdata(i, j, k, 1:4)
        end do
      end do
    end do

    deallocate(rdata, stat=err)
    if (err .NE. 0) then
      print *, "rdata array deallocation: failed"
      stop
    else
      print *, "rdata array deallocation: succeeded"
    end if

!   Terminate access to the dataspace in file.
    CALL h5sclose_f(dspace_id, err)
    if (err .EQ. -1) then
      print *, "h5 closing dataspace: failed"
    else
      print *, "h5 closing dataspace: succeeded"
```

```
      end if

!     Terminate dataset creation property list.
      CALL h5pclose_f(dset_prop_id, err)
      if (err .EQ. -1) then
        print *, "h5 closing property list: failed"
      else
        print *, "h5 closing property list: succeeded"
      end if

!     Terminate access to the array datatypes in file and memory.
      CALL h5tclose_f(dtype_id, err)
      if (err .EQ. -1) then
        print *, "h5 closing array-f datatype: failed"
      else
        print *, "h5 closing array-f datatype: succeeded"
      end if

      CALL h5tclose_f(dtypemem_id, err)
      if (err .EQ. -1) then
        print *, "h5 closing array-m datatype: failed"
      else
        print *, "h5 closing array-m datatype: succeeded"
      end if

!     Terminate access to the dataset and release resources used by it.
      CALL h5dclose_f(dset_id, err)
      if (err .EQ. -1) then
        print *, "h5 closing dataset: failed"
      else
        print *, "h5 closing dataset: succeeded"
      end if

!     Terminate access to "/Group"
      CALL h5gclose_f(group_id, err)
      if (err .EQ. -1) then
        print *, "h5 closing group: failed"
      else
        print *, "h5 closing group: succeeded"
      end if

!     Terminate access to the file.
      CALL h5fclose_f(file_id, err)
      if (err .EQ. -1) then
        print *, "h5 closing file: failed"
```

```
        else
          print *, "h5 closing file: succeeded"
        end if

!     Close FORTRAN interface.
      CALL h5close_f(err)
        if (err .EQ. -1) then
          print *, "h5 closing fortran interface: failed"
        else
          print *, "h5 closing fortran interface: succeeded"
        end if

       end if

#endif

! END DATA OUTPUT
```

New Point Plotting Utility *plotPoints.F90*

```
      PROGRAM PLOTPOINTS

      USE HDF5
      IMPLICIT NONE

      INTEGER, EXTERNAL :: get_valency, exists
      CHARACTER(5), EXTERNAL :: to_str, to_zstr

!     Define x, y, z coordinate ranges and number
!     of EM-field values to store
      INTEGER, PARAMETER :: xmin=1, xmax=10
      INTEGER, PARAMETER :: ymin=1, ymax=10
      INTEGER, PARAMETER :: zmin=1, zmax=10
      INTEGER, PARAMETER :: nf=4

      CHARACTER(64) :: point_file="points", param_file="params"
      CHARACTER(64) :: filename
      CHARACTER :: space
      CHARACTER(7), PARAMETER :: groupname = "MyGroup"    ! Group name
      CHARACTER(9), PARAMETER :: dsetname = "MyDataset" ! Dataset name

      INTEGER(HID_T) :: file_id       ! File identifier
      INTEGER(HID_T) :: group_id      ! Group identifier
      INTEGER(HID_T) :: dset_id       ! Dataset identifier
      INTEGER(HID_T) :: dspace_id     ! Dataspace identifier
      INTEGER(HID_T) :: dspacemem_id ! Memory dataspace identifier
      INTEGER(HID_T) :: dtype_id      ! Array datatype identifier
      INTEGER(HID_T) :: dtypemem_id  ! Memory array datatype identifier

!     Dataset dimension sizes and rank
      INTEGER(HSIZE_T) :: dims(3) = (/xmax-xmin+1, &
                                      ymax-ymin+1, &
                                      zmax-zmin+1/)

      INTEGER, PARAMETER :: drank = 3

!     Individual point selection parameters
!     Number of points selected
      INTEGER :: pts
```

```
!     Point coordinate array
      INTEGER, ALLOCATABLE, DIMENSION(:,:) :: co

!     CPU ranks with coordinate ranges
      INTEGER, ALLOCATABLE, DIMENSION(:,:) :: ranges

!     Distinct CPU ranks
      INTEGER, ALLOCATABLE, DIMENSION(:) :: ranks

!     Plot file names
      CHARACTER(64), ALLOCATABLE, DIMENSION(:) :: plot_file

!     Array datatype's dimension and rank
      INTEGER(HSIZE_T) :: dtype_dim(1) = nf
      INTEGER :: dtype_rank = 1

!     Read data buffer
      REAL, DIMENSION(xmax-xmin+1, ymax-ymin+1, zmax-zmin+1, nf) ::
          rdata

      INTEGER :: err              ! Error flag
      INTEGER :: i,j,r,p          ! Loop counters
      INTEGER :: t, t_init, t_end ! Timestep range
      INTEGER :: np               ! Number of processors
      INTEGER :: div_axis         ! Division axis (x=1, y=2, z=3)
      INTEGER :: rank = 0         ! Parallel version --> MPI rank

!================================================

!     Read simulation parameters
      open(unit=9, &
           file=param_file, &
           status="old", &
           iostat=err)

      if (err.NE.0) then
        print *, "Error opening '", param_file, "' for reading"
        stop
      endif

      read(9,*,iostat=err) space, t_init, t_end
      print "(A, I5, A, I5)", "t_init: ", t_init, ", t_end: ", t_end

      read(9,*,iostat=err) space, np
      print "(A, I5)", "np: ", np
```

```
      allocate( ranges(np,3), stat=err )
      if (err .NE. 0) then
        print *, "'ranges' array allocation: failed"
        stop
      end if

      do i=1, np
        read(9,*,iostat=err) space, ranges(i,1:3)
        print "(A, 3I5)", "ranges(i,0:1): ", ranges(i,1:3)
      end do

      read(9,*,iostat=err) space, div_axis
      print "(A, I1)", "division_axis: ", div_axis

      close(unit=9)

!     Read coordinates of points to plot.
      open(unit=10, &
           file=point_file, &
           status="old", &
           iostat=err)

      if (err.NE.0) then
        print *, "Error opening '", point_file, "' for reading"
        stop
      endif

      read(10,*,iostat=err) pts

      allocate( co(pts,drank+1), stat=err )
      if (err .NE. 0) then
        print *, "'co' array allocation: failed"
        stop
      end if

      allocate( ranks(pts), stat=err )
      if (err .NE. 0) then
        print *, "'ranks' array allocation: failed"
        stop
      end if

      do i=1,pts
        ranks(i) = -1
      end do
```

```
!     Read point and determine processor rank for it.
      do i=1, pts
        read(10,*,iostat=err) co(i,2:drank+1)

        do j=1, np
          if ( (co(i,div_axis+1).GE.ranges(j,2)) .AND. (co(i,div_axis+1)
             .LE.ranges(j,3)) ) then
            co(i,1) = ranges(j,1)
            if ( exists(co(i,1),ranks,pts) == 0 ) ranks(i) = co(i,1)
            exit
          end if
        end do

        print "(A, 4I5)", "co(i,2:drank+1): ", co(i,1:drank+1)
      end do

      print *, "ranks: ", ranks(1:pts)

      close(unit=10)

!     Generate target points' data file names.
!     Create and open dat-files.
!     For parallel version: current processor rank --> dat-file name

      allocate( plot_file(pts), stat=err )
      if (err .NE. 0) then
        print *, "'plot_file' array allocation: failed"
        stop
      end if

      do i=1, pts
        plot_file(i) = trim(to_str(co(i,2)))  // "-" // &
                       trim(to_str(co(i,3)))  // "-" // &
                       trim(to_str(co(i,4)))  // "_" // &
                       to_zstr(rank)          // ".dat"

        100 format(1I5, 4G15.5)
        open(unit=10+i, &
             file=trim(plot_file(i)), &
             access="sequential", &
             form="formatted", &
             status="new", &
             iostat=err)
```

```
      if (err .NE. 0) then
        print *, "Error opening ", trim(plot_file(i)), " for writting"
        stop
      endif

    end do


!   Initialize FORTRAN interface.
    CALL h5open_f(err)

!   Repeat for each h5-file
    do r=1,pts
      if (ranks(r).NE.-1) then ! For all distinct CPU ranks
        do t=t_init,t_end

          filename = to_zstr(ranks(r)) // "_" // to_zstr(t) // ".h5"
          print *, "filename: ", filename

!         Open file.
          CALL h5fopen_f(filename, H5F_ACC_RDWR_F, file_id, err)

!         Open group.
          CALL h5gopen_f(file_id, groupname, group_id, err)

!         Open dataset.
          CALL h5dopen_f(group_id, dsetname, dset_id, err)

!         Get dataset's dataspace identifier.
          CALL h5dget_space_f(dset_id, dspace_id, err)

!         Create an array datatype to store EM-values.
          CALL h5tarray_create_f(H5T_IEEE_F32LE, dtype_rank, &
                                 dtype_dim, dtype_id, err)

          CALL h5tarray_create_f(H5T_NATIVE_REAL, dtype_rank, &
                                 dtype_dim, dtypemem_id, err)

!         Read the dataset from a file into an array rdata.
          CALL h5dread_f(dset_id, dtypemem_id, rdata, dims, err)

!         Write specific points' values into corresponding dat-file.
          do p=1, pts
            if ( co(p,1) .EQ. ranks(r) ) then
```

```
                print *, "point ", p
                print *, rdata( co(p,2), co(p,3), co(p,4), 1:nf)
                print *, ""

                write(10+p,100) t, rdata( co(p,2), co(p,3), co(p,4), 1:
                    nf)

              end if
            end do

!           Terminate access to the dataspace in file.
            CALL h5sclose_f(dspace_id, err)

!           Terminate access to the dataset and release resources used
    by it.
            CALL h5dclose_f(dset_id, err)

!           Terminate access to the file.
            CALL h5fclose_f(file_id, err)

          end do
        end if
      end do

!     Close dat-files.
      do i=1, pts
        close(unit=10+i)
      end do

!     Close FORTRAN interface.
      CALL h5close_f(err)

      END PROGRAM PLOTPOINTS

! Calculates valency of a number. Valency is the number of digits in a
! number.
! Param: n - given number, type integer

      INTEGER FUNCTION get_valency(n)
        IMPLICIT NONE

        ! Input argument declaration
        INTEGER, INTENT(IN) :: n

        get_valency = int(floor(log10(real(n)))) + 1
```

```
    END FUNCTION get_valency


! Converts an integer into string.
! Param: n - given number, type integer

    CHARACTER(5) FUNCTION to_str(n)
      IMPLICIT NONE

      ! Input argument declaration
      INTEGER, INTENT(IN) :: n

      write(to_str, "(I5)") n
      to_str = trim(adjustl(to_str))

    END FUNCTION to_str


! Converts an integer into a zero-led string.
! Param: n - given number, type integer

    CHARACTER(5) FUNCTION to_zstr(n)
      IMPLICIT NONE
      INTEGER,      EXTERNAL :: get_valency
      CHARACTER(5), EXTERNAL :: to_str

      ! Input argument declaration
      INTEGER, INTENT(IN) :: n

      to_zstr = "00000"
      to_zstr(5-get_valency(n)+1:5) = trim(to_str(n))

    END FUNCTION to_zstr


! Checks whether an element exists in a set.
! Param: l - given element, type integer
!        set - set of elements, type 1-d integer array
!        upper - upper bound of set

    INTEGER FUNCTION exists(l,set,upper)
      IMPLICIT NONE

      ! Input argument declaration
```

```
    INTEGER, INTENT(IN) :: l, upper
    INTEGER, INTENT(IN) :: set(upper)

    INTEGER :: i

    exists = 0

    do i = 1,upper
      if (l .EQ. set(i)) then
        exists = 1
        exit
      end if
    end do

  END FUNCTION exists
```

Original Plane Visualisation Utility *visualisePlane.csh*

```
#!/bin/csh

set aftermax = 5

set fixlocation = $argv[1]
set CPU_ID = $argv[2]
set prefix=/home/maksims/workspace/data

# plaintosee = 1 for x-fixed plain
# plaintosee = 2 for y-fixed plain
# plaintosee = 3 for z-fixed plain

set plaintosee = 3
set seecolumn = 4
set xaxis = 2
set yaxis = 1

mkdir $fixlocation
cd $fixlocation
echo "" > empty

# xaxis in the figure displays y->2,z->3,x->1
# yaxis in the figure displays y->2,z->3,x->1

set k = 1
while ($k <  10)
set maxvalue = 100000000000
cat $prefix/E_field_0000"$k"_$CPU_ID.out | \
awk '{print $1, $2, $3, $4/1e18}' | \
gawk -v fixlocation=$fixlocation \
     -v plaintosee=$plaintosee \
     -v seecolumn=$seecolumn \
     -v xaxis=$xaxis \
     -v yaxis=$yaxis \
'($plaintosee == fixlocation){print $xaxis,$yaxis,(($seecolumn*
   $seecolumn))}'>test
set xrangemax =  `cat test | sort -n -k 1 | tail -1 | awk  '{print $1
   }'`
set xrangemin =  `cat test | sort -nr -k 1 | tail -1 | awk  '{print $1
   }'`
@ xrange =  1 + $xrangemax - $xrangemin
```

```
set yrangemax = `cat test | sort -n -k 2 | tail -1 | awk '{print $2
    }'`
set yrangemin = `cat test | sort -nr -k 2 | tail -1 | awk '{print $2
    }'`
@ yrange = 1 + $yrangemax - $yrangemin

set min = `cat test | sort -gr -k 3 | tail -1 | awk '{print $3}'`
set max = `cat test | sort -g -k 3 | tail -1 | awk '{print $3}'`
set range = `cat empty|awk -v min=$min -v max=$max '{print max - min}'`
echo $range, $min,$maxvalue
cat test | awk -v range=$range -v min=$min -v maxvalue=$maxvalue \
'{print $1,$2,int((($3-min)/range)**0.2*maxvalue)**0.5)}' > test2
echo "P2" > test.pgm
echo "# test" >> test.pgm
echo $xrange $yrange >> test.pgm
echo ""| awk -v maxvalue=$maxvalue '{print int((maxvalue)**0.5)}' >>
    test.pgm
cat test2 | awk '{print $3}' >> test.pgm
cat test.pgm | \
awk '((NR > 4)&&($1 < 1)){print 0 }((NR > 4)&& ($1 > 0)){print $0}(NR <
     5){print $0}' > testtest.pgm
mv testtest.pgm test.pgm
/home/maksims/workspace/src/post/pgm2ppm
cat test.ppm | awk '(NR == 1){print $1}(NR > 1){print $0}' > test$k.ppm
@ k ++
end

#================================================

set k = 10
while ($k < 100)
set maxvalue = 1000
cat $prefix/E_field_000"$k"_$CPU_ID.out | \
awk '{print $1, $2, $3,$4/1e18}' | \
gawk -v fixlocation=$fixlocation \
     -v plaintosee=$plaintosee \
     -v seecolumn=$seecolumn \
     -v xaxis=$xaxis \
     -v yaxis=$yaxis \
'($plaintosee == fixlocation){print $xaxis,$yaxis,(($seecolumn*
   $seecolumn)) }' > test
set xrangemax = `cat test | sort -n -k 1 | tail -1 | awk '{print $1
    }'`
```

```
set xrangemin = `cat test | sort -nr -k 1 | tail -1 | awk '{print $1
    }'`
@ xrange = 1 + $xrangemax - $xrangemin

set yrangemax = `cat test | sort -n -k 2 | tail -1 | awk '{print $2
    }'`
set yrangemin = `cat test | sort -nr -k 2 | tail -1 | awk '{print $2
    }'`
@ yrange = 1 + $yrangemax - $yrangemin

set min = `cat test | sort -gr -k 3 | tail -1 | awk '{print $3}'`
set max = `cat test | sort -g -k 3 | tail -1 | awk '{print $3}'`
set range = `cat empty|awk -v min=$min -v max=$max '{print max - min}'`
cat test | awk -v range=$range -v min=$min -v maxvalue=$maxvalue \
'{print $1,$2,int((((($3-min)/range)**0.2*maxvalue)**0.5)}' > test2
echo "P2" > test.pgm
echo "# test" >> test.pgm
echo $xrange $yrange >> test.pgm
echo ""| awk -v maxvalue=$maxvalue '{print int((maxvalue)**0.5)}' >>
    test.pgm

cat test2 | awk '{print $3}' >> test.pgm
cat test.pgm | awk '((NR > 4)&&($1 < 1)){print 0 }((NR > 4)&& ($1 > 0))
    {print $0}(NR < 5){print $0}' > testtest.pgm
mv testtest.pgm test.pgm
/home/maksims/workspace/src/post/pgm2ppm
cat test.ppm | awk '(NR == 1){print $1}(NR > 1){print $0}' > test$k.ppm
@ k ++
end

#================================================

set k = 100
while ($k < 1000)
set maxvalue = 1000

cat $prefix/E_field_00"$k"_$CPU_ID.out | \
awk '{print $1,$2,$3,$4/1e18}' | \
gawk -v fixlocation=$fixlocation \
     -v plaintosee=$plaintosee \
     -v seecolumn=$seecolumn \
     -v xaxis=$xaxis \
     -v yaxis=$yaxis \
'($plaintosee == fixlocation){print $xaxis,$yaxis,(($seecolumn*
    $seecolumn)) }' > test
```

```
set xrangemax = `cat test | sort -n -k 1 | tail -1 | awk '{print $1
    }'`
set xrangemin = `cat test | sort -nr -k 1 | tail -1 | awk '{print $1
    }'`
@ xrange = 1 + $xrangemax - $xrangemin

set yrangemax = `cat test | sort -n -k 2 | tail -1 | awk '{print $2
    }'`
set yrangemin = `cat test | sort -nr -k 2 | tail -1 | awk '{print $2
    }'`
@ yrange = 1 + $yrangemax - $yrangemin

set min = `cat test | sort -gr -k 3 | tail -1 | awk '{print $3}'`
set max = `cat test | sort -g -k 3 | tail -1 | awk '{print $3}'`
set range = `cat empty|awk -v min=$min -v max=$max '{print max - min}'`
cat test | awk -v range=$range -v min=$min -v maxvalue=$maxvalue \
'{print $1,$2,int((((($3-min)/range)**0.2*maxvalue)**0.5)}' > test2
echo "P2" > test.pgm
echo "# test" >> test.pgm
echo $xrange $yrange >> test.pgm
echo ""| awk -v maxvalue=$maxvalue '{print int((maxvalue)**0.5)}' >>
    test.pgm
cat test2 | awk '{print $3}' >> test.pgm
cat test.pgm | awk '((NR > 4)&&($1 < 1)){print 0 }((NR > 4)&& ($1 > 0))
   {print $0}(NR < 5){print $0}' > testtest.pgm
mv testtest.pgm test.pgm
/home/maksims/workspace/src/post/pgm2ppm
cat test.ppm | awk '(NR == 1){print $1}(NR > 1){print $0}' > test$k.ppm
@ k ++
end

#================================================

set k = 1000
while ($k < 5000)
set maxvalue = 1000

cat $prefix/E_field_0"$k"_$CPU_ID.out | awk '{print $1,$2,$3,$4/1e18}'
   | \
gawk -v fixlocation=$fixlocation \
     -v plaintosee=$plaintosee \
     -v seecolumn=$seecolumn \
     -v xaxis=$xaxis \
     -v yaxis=$yaxis \
```

```
'($plaintosee == fixlocation){print $xaxis,$yaxis,(($seecolumn*
   $seecolumn)) }' > test
set xrangemax = `cat test | sort -n -k 1 | tail -1 | awk '{print $1
   }'`
set xrangemin = `cat test | sort -nr -k 1 | tail -1 | awk '{print $1
   }'`
@ xrange = 1 + $xrangemax - $xrangemin

set yrangemax = `cat test | sort -n -k 2 | tail -1 | awk '{print $2
   }'`
set yrangemin = `cat test | sort -nr -k 2 | tail -1 | awk '{print $2
   }'`
@ yrange = 1 + $yrangemax - $yrangemin

set min = `cat test | sort -gr -k 3 | tail -1 | awk '{print $3}'`
set max = `cat test | sort -g -k 3 | tail -1 | awk '{print $3}'`
set range = `cat empty|awk -v min=$min -v max=$max '{print max - min}'`
cat test | awk -v range=$range -v min=$min -v maxvalue=$maxvalue \
'{print $1,$2,int((((($3-min)/range)**0.2*maxvalue)**0.5)}' > test2
echo "P2" > test.pgm
echo "# test" >> test.pgm
echo $xrange $yrange >> test.pgm
echo ""| awk -v maxvalue=$maxvalue '{print int((maxvalue)**0.5)}' >>
    test.pgm

cat test2 | awk '{print $3}' >> test.pgm
cat test.pgm | awk '((NR > 4)&&($1 < 1)){print 0 }((NR > 4)&& ($1 > 0))
   {print $0}(NR < 5){print $0}' > testtest.pgm
mv testtest.pgm test.pgm
/home/maksims/workspace/src/post/pgm2ppm
cat test.ppm | awk '(NR == 1){print $1}(NR > 1){print $0}' > test$k.ppm
@ k ++
end

exit 0
```

Compilation Script for Parallel HDF5 Programs

h5compile.sh

```
#!/bin/bash

fullname=$1
basename=`basename $fullname .F90`

h5pfc -c $fullname
h5pfc -o $basename $basename.o

exit 0
```

Bibliography

[Bér94] Jean-Pierre Bérenger. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 114(2):185–200, 1994.

[Cos05] Fumie Costen. *High Speed Computational Modelling in the Application of UWB Signals*. PhD thesis, University of Kyoto, Japan, 2005.

[Cos06] João Costa. Application of high performance computing to FDTD for UWB systems, September 2006.

[CYCA03] Christian M. Chilan, MuQun Yang, Albert Cheng, and Leon Arber. Parallel I/O performance study with HDF5, a scientific data package. Technical report, The National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, 2003.

[EPL94] T. M. R. Ellis, Ivor R. Phillips, and Thomas M. Lahey. *Fortran 90 Programming*. Addison-Wesley, 1st edition, May 1994.

[Fol02] Mike Folk. Introduction to HDF5, presentation slides. Technical report, The National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, 2002.

[GBOM04] S. González García, A. Rubio Bretones, B. García Olmedo, and R. Gómez Martín. *Time Domain Techniques in Computational Electromagnetics*. WIT Press / Computational Mechanics, 2004.

[GLNS+05] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific data management in the coming decade. *ACM SIGMOD Record*, 34(4):34–41, January 2005.

[GMSS07] Lakner Gary, Gary L. Mullen-Schultz, and Carlos Sosa. *IBM System Blue Gene Solution: Application Development*. IBM International Technical Support Organization, June 2007.

[Gro02] The HDF5 Group. HDF5 wins 2002 R & D 100 award. Technical report, The National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, 2002.

[Gro04] The HDF5 Group. Performance evaluation report: gzip, bzip2 compression with and without shuffling algorithm. Technical report, The National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, 2004.

[Gro07] The HDF5 Group. *HDF5 Online Tutorial*. University of Illinois at Urbana-Champaign, 2007. http://hdf.ncsa.uiuc.edu/HDF5/Tutor/ Accessed on May 15, 2007.

[Lan04] Mark Lancaster. Building programs: Compiling, linking, running, presentation slides. Technical report, May 2004.

[Pou02] Elena Pourmal. FITSIO, HDF4, NetCDF, PDB and HDF5 performance, some benchmarks results. *Science Data Processing Workshop*, February 2002.

[RD90] Russ Rew and Glenn Davis. NetCDF: an interface for scientific data access. *Computer Graphics and Applications, IEEE*, 10(4):76–82, 1990.

[SHS07] Jürgen Schnack, Peter Hage, and Heinz-Jürgen Schmidt. Optimized implementation of the Lanczos method for magnetic systems. *Journal of Computational Physics*, 2007.

[Smi95] Ian Moffat Smith. *Programming in Fortran 90: A First Course for Engineers and Scientists*. John Wiley & sons, April 1995.

[SS95] Kurt L. Shlager and John B. Schneider. A survey of the Finite-Difference Time-Domain literature. *Antennas and Propagation Magazine, IEEE*, 37(4):39–57, 1995.

[Thi06] Arnaud Thiry. *Efficient FDTD for Broadband Systems*. PhD thesis, University of Manchester, School of Computer Science, Kilburn Building, Manchester M13 9PL, United Kingdom, May 2006.

[VAN99] Scott Vetter, Yukiya Aoyama, and Jun Nakano. *RS/6000 SP: Practical MPI Programming*. IBM International Technical Support Organization, August 1999.

[Yee66] Kane S. Yee. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *Antennas and Propagation, IEEE Transactions on [legacy, pre-1988]*, 14(3):302–307, 1966.

[Zha07] Yongwei Zhang. Phased array antenna element design, presentation slides. Technical report, School of Electrical and Electronic Engineering, University of Manchester, 2007.