|epcc|

Poisson Solvers for Electrokinetic Problems

Ruairi Short

August 23, 2013

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2013

Abstract

This report sets out to investigate the use of Fast Fourier Transforms (FFTs) to solve Poisson's equation in the lattice Boltzmann code Ludwig. The parallel 3D FFT library P3DFFT was used to conduct the FFTs, though it was found to be difficult to integrate with the current code. An algorithm to switch between Cartesian and pencil decompositions was written to combine this library and Ludwig. Results were obtained from tests conducted against the old successive over-relaxation (SOR) solving method and theoretical solutions to the Gouy Chapman and liquid junction problems to provide evidence that the FFT method was consistent. Timing results on HECToR phase 3 and BlueGene/Q also showed the FFT method to solve Poisson's equation approximately 100 times faster than the SOR solver, with BlueGene/Q showing excellent scaling up to 4096 cores.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank both of my project supervisors, Dr. Kevin Stratford and Dr. Oliver Henrich. They were extremely helpful throughout the course of this project, providing me with guidance and assistance in the direction of the project.

Chapter 1

Introduction

Poisson's equation is of fundamental importance in computational physics. The equation appears in Newtonian gravity (specifically Gauss' Law for gravity), electromagnetic and hydrodynamic problems. These cover a very wide range of applications. For example, in an electrostatics problem with electric charges in a system, the solution to Poisson's equation describes the electric potential for a given charge distribution[16]. The electric potential must be found in order to determine the forces acting on particles in the system, so that their positions can be updated.

Poisson's equation is an elliptical partial differential equation. As such it is not, in general, analytically solvable. A number of different methods are employed to solve it numerically. These include Fourier methods[31], multigrid methods[34] and particle-mesh methods[19], each with varying degrees of speed and complexity.

This project is concerned with Poisson's equation in a piece of code called Ludwig. In its most general form, Ludwig is a "parallel lattice Boltzmann code for complex fluids". The term complex fluids covers a large number of materials but can be described in general as "[f]luids showing non-linear viscous behaviours, as well as viscoelastic materials"[7]. Everyday examples include ice-cream and shaving foam, though these are not usually modelled in Ludwig. It was started by Jean-Cristophe Desplat of the Edinburgh Centre for Parallel Computing, along with Ignacio Pagonabarraga and Peter Bladon of the Department of Physics and Astronomy in the University of Edinburgh[11]. Now, it is updated and maintained by a much larger team in many different locations. The purpose of Ludwig is to provide a framework for future users to build upon. In reality, it is a set of codes, that use a set of common routines, such as communications and I/O. This allows the user to concentrate on the physics they wish to model and not have to worry about the parallel computing part. Development is still active and ongoing. The main code was recently ported to GPUs on Titan[33], the world's largest supercomputer at the time, in a bid to see performance improvements[14].

Work is currently active on a new electro-kinetics module, and this is where Poisson's equation appears. This new module will simulate systems that contain electric charges which are able to freely move about the system. For example modelling what occurs

when two liquids containing electric charges are brought into contact. In order to update the position of these charges, it is necessary to know the electric potential in the system, and this is found by applying Poisson's equation.

In the first incarnation of this electro-kinetics module, successive over-relaxation (SOR) was used to solve Poisson's equation. While the physics being modelled was correct, and the implementation was relatively easy, the performance seen was quite poor. The scaling to large numbers of cores was cited as the main problem[24] with this algorithm. Obviously this poses issues for large three dimensional problems, where one would like to use large numbers of cores, and where the most interesting physics is inevitably found.

Another solution was necessary and Fast-Fourier Transforms (FFTs) were the next logical step as a method to solve Poisson's equation. Faster methods such as multigrid were deemed too complex to implement in the short amount of time that was available for this project.

This project sets out to determine if FFTs are a better method of solving Poisson's equation, through the use of a parallel FFT library. Two main architectures were used for this, HECToR and BlueGene/Q, in order to investigate performance on both and to see how the implementation might translate between systems.

1.1 Report Structure

This report begins by introducing the background necessary to understand Poisson's equation. A discussion of the problems with the SOR solver is followed by an explanation of FFTs and how they can be used to solve Poisson's equation. Also provided is a description of the methods of improving the performance of 3D FFTs by using a pencil decomposition. In order to ensure the FFT solver is working correctly, some accuracy tests will be necessary and the theory behind these is also provided.

The second chapter is an outline of the design and implementation, with details of the main target architectures and compilation methods. Here, the specifics of implementing an FFT solver for Poisson's equation in Ludwig are also outlined, including the installation of the library used.

Next, the results of the accuracy tests are presented followed by a comparison of the SOR solver and the FFT solver. These are analysed and discussed with graphs used to show the details.

Finally, the impact of using an FFT solver instead of an SOR solver will be discussed in the conclusions, along with suggestions of further improvements that could be made.

Chapter 2

Background Information and Theory

2.1 Ludwig

Ludwig is a "parallel lattice Boltzmann code for complex fluids"[11]. As a project, it is current with many different branches. Initially the focus was on binary fluid mixtures with and without solids present[11]. This has since expanded to include colloidal suspensions and liquid crystal flows. It has been used for some interesting results, including binary fluid simulations and colloidal suspensions For examples see [22] and [6]. For the purposes of this project we will be working in the electro-kinetics module. As a general overview of the operation of the code, the main steps of an execution of are outlined below:

- Initialisation: read input files and set up domain. Electric charges are placed throughout the system according to the problem to be modelled. Then the following are iterated over:
- Field Computation: $\rho(\vec{x})$, the electric field at all points, is computed.
- Solve Poisson's equation: This gives the potential Ψ for the system.
- Solve the Nernst Planck equation: This updates the positions of the charged fluid.

This allows for systems containing electric charges, where the electric charge carriers can move, to be modelled. An everyday example of a problem this can be used to model is an oil and water mixture, in the presence of electric fields. This particular problem can produce some interesting results as can be seen in the paper by Rotenberg et al.[32].

2.1.1 Ludwig Internals

Ludwig splits the problem volume into lattice sites, the number of which is specified by the user at runtime, giving the problem size. Every lattice site has integer value coordinates corresponding to its location in the lattice. This can be thought of as each

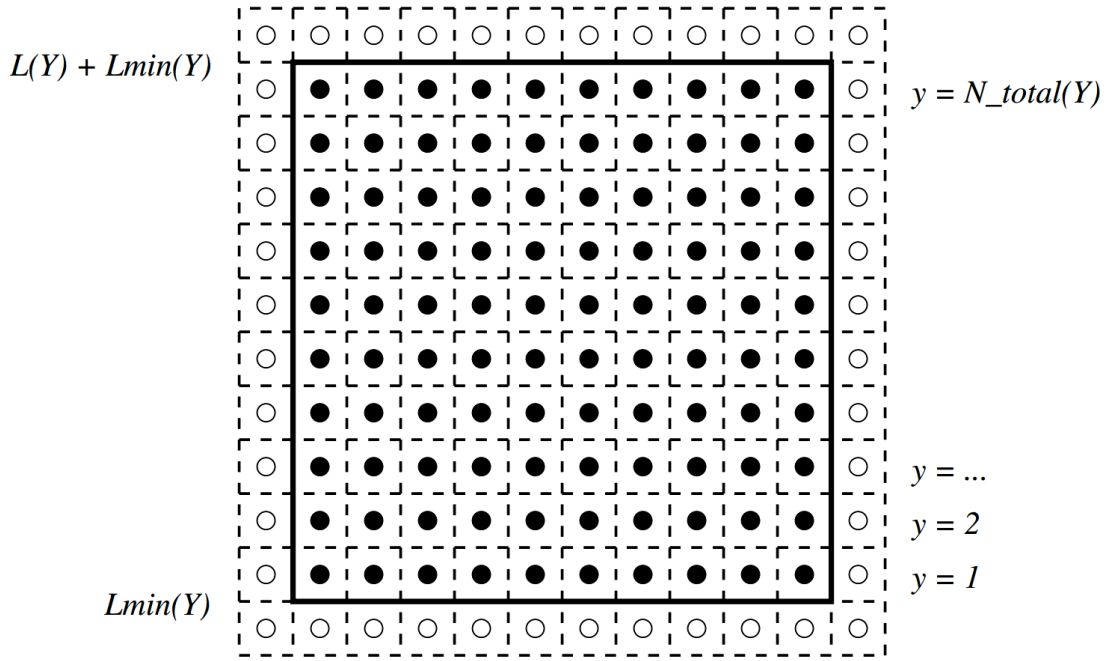


Figure 2.1: Diagram of the lattice used in Ludwig in two dimensions. Lattice sites are located at $y = 1, \ldots, N$. Here lattice sites are represented by full circles and halo sites by open circles. The third dimension is added by stacking many of these grids on top of each other and the inclusion of halos at the ends. The lattice sites will then be located at the centre of a volume. Reproduced from [24]

lattice point occupying the centre of a control volume one lattice unit on a side. The effect of this is that the edges of the system have an offset of half a lattice unit from the lattice points themselves and this must be remembered, in particular when plotting graphs.

The code provides a number of environments to allow ease of use. The parallel environment abstracts the basic environment. This allows the code to be used in serial or parallel, by returning the appropriate values depending on the number of processors. Parallelism is introduced through the use of message passing. When run in parallel this environment supports communications in `MPI_COMM_WORLD`. A coordinate environment is provided that allows external routines access to the Cartesian coordinates that represent the system. This includes routines that provide access to the MPI Cartesian communicator which handles most of the communication. Also included in this module, that will be of use for this project, are routines to access the total system size, the Cartesian coordinates of the calling process in the Cartesian communicator, and the global coordinates of the first element of the local array.

In order to allow the user greater control over the simulations to run, Ludwig uses an input file to intitialise many parameters in the system. This uses key value pairs which are maintained throughout execution and the values can be retrieved by the user. As a feature, this should prove useful for running tests, as different input files can be specified

without the need for recompiling between each run.

In order to parallelise the computation, Ludwig uses a domain decomposition method. These domains are normally chosen to keep the load as balanced as possible and minimise the size of the boundaries, as the SOR solver requires halos to be swapped. Since there are never holes in the system to be modelled this means that a three dimensional Cartesian processor decomposition is used with the sizes of each dimension as close as possible. No process ever holds all the data. Processors initialise the data independently at start-up. This improves the efficiency of the program by reducing the memory footprint of the program as a whole and reducing the amount of communication necessary. It is also possible for the user to specify a different decomposition by changing the appropriate values in the input file before execution.

Typically problems are 64^3 or larger, but not more than 256^3, as this is large enough to produce interesting results while larger problems would take excessively long to run. It is important to note that the problems are solved using periodic boundary conditions, which also gives the effect of a larger simulation volume.

In previous tests the current method of solving Poisson's equation, by successive over-relaxation (SOR), was shown to be slow and scaling poorly as can be seen in figure 2.2. Here, a problem size of 128^3 was being solved on the Mare-Nostrum supercomputer

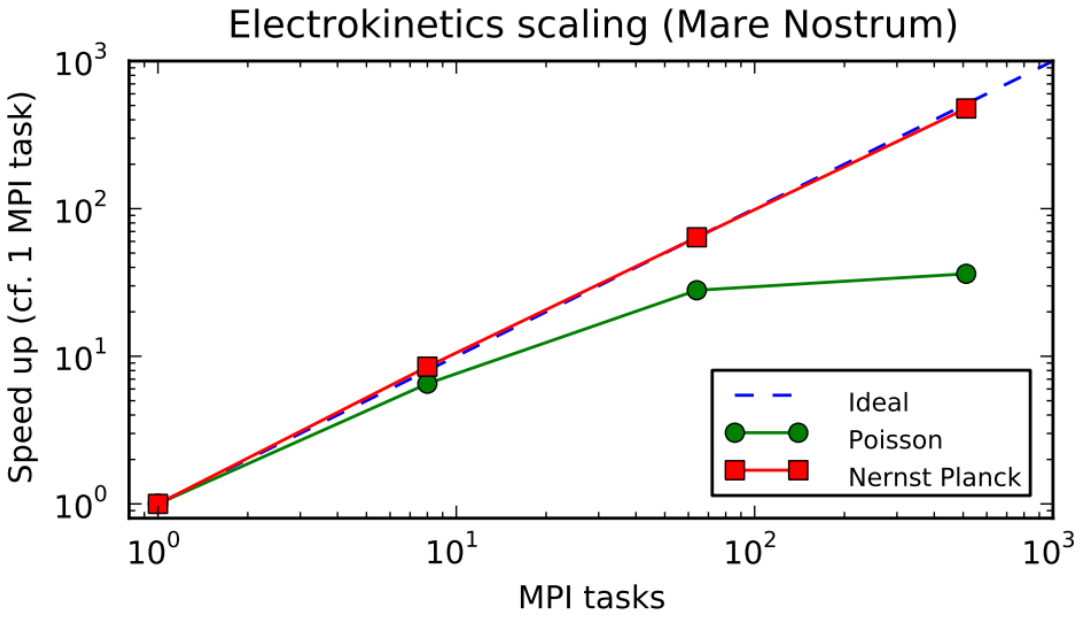


Figure 2.2: Graph of Speedup against MPI tasks for the current SOR solver in Ludwig. Mare-Nostrum is a supercomputer in the Barcelona Supercomputing Centre. This shows the poor performance of the SOR solver compared with the other main time consuming piece of the code, the Nernst-Planck equation. Reproduced from [24]

in the Barcelona Supercomputing Centre. This graph shows also shows the scaling of the part of the code dealing with solving the Nernst-Planck equation. Looking at the

absolute times on two processors, one sees that for this problem size solving the Nerst-Planck equation takes 1.07 s while solving Poisson's equation takes 19.8 s[1]. These are the two major time consuming parts of the electro-kinetics module of Ludwig. As can be seen the SOR method of solving Poisson's equation has much worse performance as the MPI tasks are increased. For this reason it was seen as a bottleneck preventing larger problems being modelling in reasonable times and has thus become the focus of this project.

2.2 Successive Over Relaxation

As a method for solving PDEs, SOR is an old one, having been fairly standard practice in the 1970s[31]. It is an improvement on the Gauss-Seidel algorithm, through the use of over-relaxation and Chebyshev acceleration. For a full discussion the reader is referred to section 19.5 of Numerical Recipes in C[31].

When dealing with electrostatics, as we are in Ludwig, Poisson's equation takes the form

$$\frac{\partial \Psi(\vec{x})^2}{\partial^2 x} = -\rho(\vec{x})/\epsilon. \tag{2.1}$$

Here, Ψ is the electric field and ρ is the electric charge at each x. Only problems where ϵ, the dielectric permittivity, does not vary with x will be dealt with. The case where it varies is significantly more complex. The SOR update equation then takes the form:

$$\Psi^{n+1}_{i,j,k} = \Psi^{n}_{i,j,k} - \frac{\delta t \rho_{i,j,k}}{\epsilon}. \tag{2.2}$$

Ψ is found by iterating over the lattice sites and through solutions to this equation, choosing an optimum δt in order to improve the rate of convergence. Ludwig further improves on the rate of convergence by updating δt at each iteration through the use of Chebyshev acceleration. Unfortunately, while easy to program, this method is still inefficient when it comes to large problems[31].

In order to compute the solution, each lattice site must be iterated over, resulting in 6 operations per site. It is an iterative refinement scheme, so this process must be repeated until the error is acceptable. Optimally, this method takes N iterations to converge for an $N \times N \times N$ grid[31]. However, the optimum is rarely attained. The total number of operations is thus of O(N^4) for three dimensional problems. To be more exact, assuming N iterations the number of operations is given by:

$$\text{Complexity} = 6 \times N^3 * N \tag{2.3}$$

This method does not match what should be attainable with a different algorithm, as methods with computational complexity of O(N) exist.

[1] On HECToR for the liquid junction problem which will be discussed later

2.3 Solving Poisson's equation with Fast Fourier Transforms

Three main further options were noted to be possible to implement; Fast Fourier Transforms(FFTs), the Fast Multipole Method(FMM) and Multigrid methods. Of these, FFTs are the most well known and thus well documented and used. Libraries exist for solving FFTs whereas the other methods would need to be implemented entirely from the beginning. They are all complex to code with much thought necessary in order to achieve the best performance, which is critical in this project. Due to the relatively short time available it was decided that FFTs would be the best solution to use.

The method of solving Poisson's equation with FFTs is relatively easy to derive. We shall present it in 1D, and the 3D solution is easily obtainable from this.

The general form of the Fourier Transform is given by

$$\theta(k) = \frac{1}{\sqrt{2\pi}} \int_{\infty}^{-\infty} \Psi(x) e^{-ikx} dx \tag{2.4}$$

where $\theta(k)$ is known as the Fourier transform of $\Psi(x)$. The set of coordinates k, is often called Fourier space, while the set of coordinates x is usually referred to as real space. The inverse operation is

$$\Psi(x) = \frac{1}{\sqrt{2\pi}} \int_{\infty}^{-\infty} \theta(k) e^{ikx} dk. \tag{2.5}$$

If we call σ the Fourier transform of ρ, we can rewrite Poisson's equation as follows:

$$\frac{1}{\sqrt{2\pi}} \int_{\infty}^{-\infty} \theta(k) \frac{\partial^2}{\partial^2 x} e^{ikx} dk = -\frac{1}{\epsilon\sqrt{2\pi}} \int_{\infty}^{-\infty} \sigma(k) e^{ikx} dk. \tag{2.6}$$

Here, we have brought the second derivative of x inside the integral as the integral is over k. Also, since ϵ has no x dependence, we could bring it outside of the integral on the right hand side. Cancelling the $\frac{1}{\sqrt{2\pi}}$ from each side is the next step. Then conduct the differentiation on the left hand side, giving

$$\int_{\infty}^{-\infty} -k^2 \theta(k) e^{ikx} dk = -\frac{1}{\epsilon} \int_{\infty}^{-\infty} \sigma(k) e^{ikx} dk. \tag{2.7}$$

The next step is not so obvious. In order for the integrals to be equal, the integrands must be equal since the integral is over the same variable with the same limits. This allows us to remove the integrals and multiply across by $-e^{ikx}$, leaving us with

$$k^2 \theta(k) = \frac{1}{\epsilon} \sigma(k). \tag{2.8}$$

Since we ultimately want to solve for θ,

$$\theta(k) = \frac{1}{k^2 \epsilon} \sigma(k). \tag{2.9}$$

Ψ is then found using equation 2.5. Thus, solving Poisson's equation using Fourier transforms can be broken down into three steps.

- Compute σ, the Fourier transform of ρ.
- Multiply by $\frac{1}{k^2\epsilon}$ in Fourier space, giving θ.
- Compute the inverse Fourier transform of θ, resulting in Ψ.

The generalization of this method to three dimensions is simple. Fourier transforms are just integrals so in three dimensions there are integrals over the x, y and z directions. There are no interdependencies, each can simply be performed in turn.

2.3.1 Fourier Transforms

In Ludwig, the system is split into a lattice. As such, discrete Fourier transforms will need to be used. This is only natural in computational scenarios. Discrete Fourier Transforms estimate the Fourier Transform of a function by sampling a finite number of its points. If we have a domain such that $0 \leq x \leq L$ in one dimension, we can define a lattice of N equally spaced points such that $x_n = \frac{nL}{N}, n = 0, \ldots, N-1$. The function is sampled at each of these points. The discrete Fourier coefficients are then given by

$$\theta(k) = \sum_{n=0}^{N-1} exp(-\frac{2ink\pi}{N})\Psi(n) \tag{2.10}$$

with inverse transformation

$$\Psi(n) = \frac{1}{N}\sum_{k=0}^{N-1} exp(\frac{2ink\pi}{N})\theta(k). \tag{2.11}$$

Computing the Fourier transform is then just computing each of these coefficients and again is easily generalisable to three dimensions in the same way as the general form. For example:

$$\Psi(l,m,n) = \frac{1}{L \times M \times N}\sum_{k_x=0}^{L-1}\sum_{k_y=0}^{M-1}\sum_{k_z=0}^{N-1} e^{\frac{2ilk_x\pi}{L}} e^{\frac{2imk_y\pi}{M}} e^{\frac{2ink_z\pi}{N}} \theta(k_x, k_y, k_z). \tag{2.12}$$

This has a computational complexity of O(N^2)[30], which is better than the SOR scheme but can be improved upon.

In the mid 1960s, J.W. Cooley and J.W.Turkey rediscovered (originally found by Gauss in 1866) a scheme for computing Fourier transforms in O($Nlog(N)$) operations[8] where N is the total number of points in the lattice. In the most simple terms, this can be achieved by repeatedly splitting the Fourier transform into the odd and even sites in the interval, until each site has been isolated. The total computational complexity for solving Poisson's equation can then be computed roughly using:

$$\text{Complexity} = 2 \times Nlog_2(N) + N. \tag{2.13}$$

We multiply by two since a forward and backward Fourier transform must be considered. The addition of N is due to the k^2 multiplication, which must take place at each lattice site. While some operations are requires to calculate k^2, this much less than N operations. Thus, this equation should closely approximate the complexity.

2.3.2 Parallel FFTs in 3D

Ordinarily when a 3-dimensional problem is decomposed among a group of processors, a 3-dimensional processor decomposition is chosen. This is often used because of the communication patterns that are most often present in parallel codes. Ludwig uses a 3-dimensional Cartesian processor decomposition as the information adjacent to each lattice point in each direction is important and this decomposition allows for minimising the communications necessary, by minimising the sizes of the boundaries between each processor.

When conducting FFTs, they are done row by row. The impact of this is that a 3D processor decomposition will not work, as each processor needs the entire row of data in order to do the FFT. 1D slab, or 2D pencil decompositions are used instead. The 2D pencil decomposition is the most desirable as it allows for a larger number of processors to be used. If the 1D slabs are used, the number of processors is limited by the extent of each dimension in the problem. For example, with a problem size of $64 \times 64 \times 64$, the maximum number of processors that can be used with 1D slabs is 64, as any dimension cannot be split into more pieces than it contains. 2D pencils will allow for up to $64 \times 64 = 4096$ processors.

There is a downside to the pencil decomposition however, more communication is necessary. With the slab decomposition 2 dimensions are kept on any processor at any given time, so the data needs to be transposed across processors only once in order to conduct the 3D FFT. When using the pencil decomposition, each processor will only have one entire dimension at any time, so two global transposes will be necessary to compute the FFT, and one more if the data is required in Fourier space in the original decomposition.

Each of these global transpositions are done using `MPI_Alltoall` calls. These calls will most likely prove to be what limits the scalability of using FFTs for this problem. `MPI_Alltoall` calls are largely limited by the bi-sectional bandwidth of the network being used for communication. This is due to the fact that each group holds half the data, and must send half of that data, which is a quarter of the total, through the bisection.

Many large HPC systems today are made up of shared memory nodes. Thus, by having the first all-to-all take place only inside a node significant performance gains can be seen as messages can be passed in shared memory. The library being used, P3DFFT, should do this to some degree, but it is important that the user specify appropriate numbers of processors for the problem.

The large number of processors that can be used make the pencil decomposition more useful. It would also be possible to use clever communicators and shared memory programming to improve performance on current shared memory nodes systems. Also, the library we will use leaves the data in transposed form in Fourier space, and returns it to it's original form upon computing the backward transform. This improves performance by reducing the number of all-to-all communications by two.

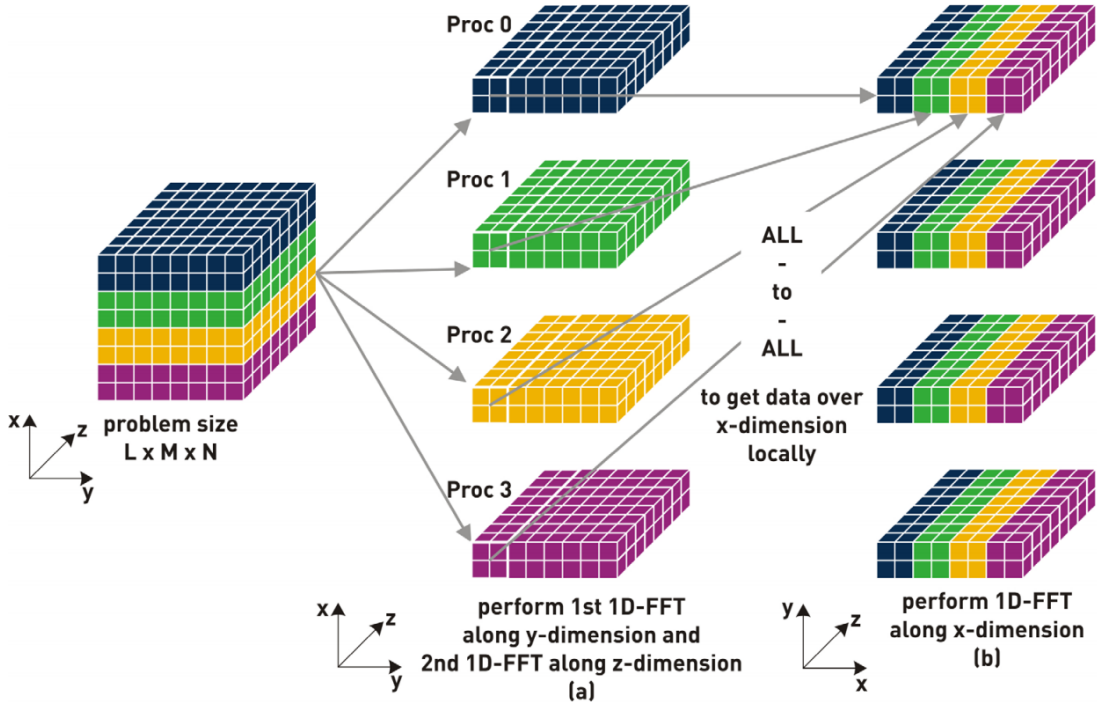


Figure 2.3: Steps involved in computing a 3D FFT with slab decomposition. Reproduced from [20]

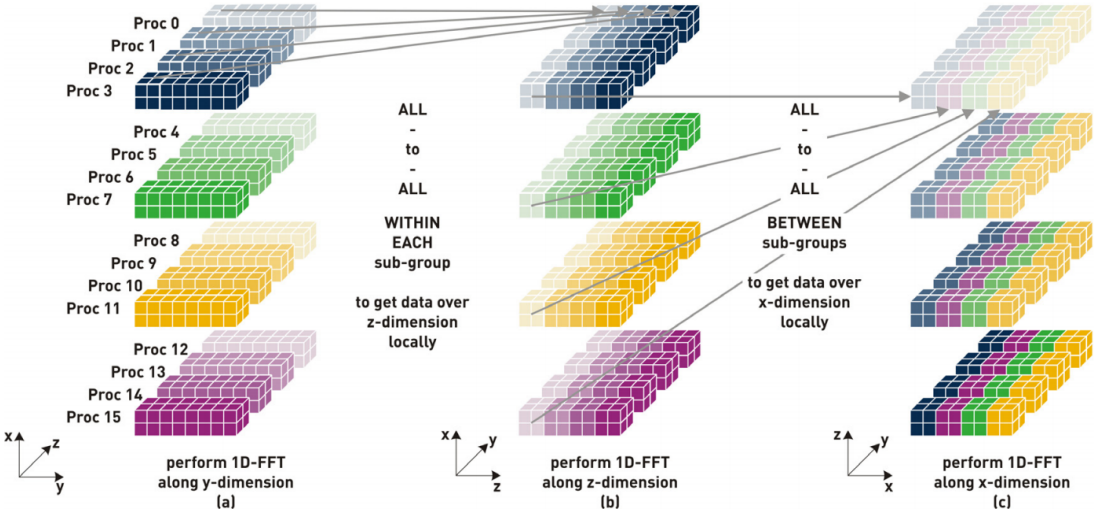


Figure 2.4: Steps involved in computing a 3D FFT with pencil decomposition. Reproduced from [20]

2.3.3 k^2 Multiplication

The last piece necessary to solve Poisson's equation is the multiplication by $\frac{1}{k^2}$. It is not immediately obvious how it can be computed from the original lattice. k^2 is given by

$$k^2 = (k_x^2 + k_y^2 + k_z^2). \tag{2.14}$$

The k_x value at each point can be calculated from

$$k_x = \frac{2\pi l}{N_x} \tag{2.15}$$

with N_x being the system size in the x-direction, with $l = 0, \ldots, N_x - 1$ representing the lattice point position, and similarly for the y and z dimensions. For the discretised system, in three dimensions, we simply compute this value at each lattice point. k_y and k_z are computed similarly. k^2 is then easily found from this.

2.4 Transforming between Processor Decompositions

As discussed earlier, the optimal processor decomposition for computing parallel 3D FFTs is a pencil decomposition. For this reason, P3DFFT expects input such that the processors are in a pencil decomposition. Since Ludwig uses a 3D Cartesian decomposition it is necessary to transform between these two decompositions before computing the FFTs.

There are two distinct stages to this, sending and receiving. The first step is to determine what data must be sent, and what process to send it to. To compute this, each processor should look at the global position of the lattice points that are in its local domain.

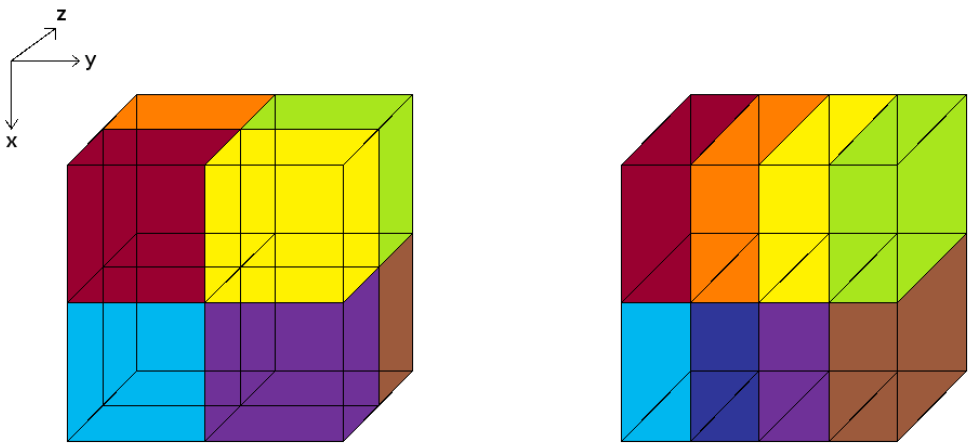


Figure 2.5: Diagram showing how 8 processors are decomposed in Cartesian (left) and pencil (right) decomposition. Colours correspond to processors with the same `MPI_Rank`.

The first thing to note is that only the non-contiguous memory directions are important to consider when looking at the location. In the pencil decomposition, each processor has a full depth (z dimension in figure 2.5) stored locally. This means that for any given

location in the z direction, with the same x and y coordinates, the destination processor is the same. Once the row and column dimensions of the pencil decomposition are known the destination processor can then be computed. We will only be solving problems with numbers of processors and decompositions where the entire domain can be divided evenly across all the processors. This means that each process can use its local knowledge of the pencil decomposition sizes. The other information that is necessary is the dimensions of the actual decomposition, i.e. the number of processors in each dimension.

The coordinates of the destination processor in the pencil decomposition can be computed using

$$pencil\_proc\_coord_x = global\_coord_x * pencil\_size_x.$$

and similarly for the y-direction. $pencil\_proc\_coord_x$ is the x coordinate of the destination processor in the pencil decomposition, $global\_coord_x$ is the position of the lattice point in the global lattice with respect to the x dimension, and $pencil\_size_x$ is the size of the local arrays in the x dimension. Finally,

$$dest\_proc = pencil\_proc\_coord_x * pencil\_size_y + pencil\_proc\_coord_y \tag{2.16}$$

can be applied to each lattice site in order to compute the destination for that element.

Following this, we must also compute processors to receive from. A similar method can be used but the depth must now be considered:

$$\begin{aligned} recv\_proc = cart\_proc\_coord_x * cart\_size_y * cart\_size_z + \\ + cart\_proc\_coord_y * cart\_size_z + cart\_proc\_coord_z. \end{aligned} \tag{2.17}$$

2.5 Tests of Accuracy

Before the FFT solver could be timed and declared fit for use, it was necessary to perform some tests to ensure the behaviour was consistent. Two main avenues for testing were identified, testing directly against the SOR solver on one solve and testing the results given by Ludwig for the Gouy-Chapman theory for electric double layers in front of a charged wall[25].

Firstly, it was desired to test the FFT solver against the SOR solver in a standalone test. This would allow for problems to be corrected quickly if they were encountered. In order to obtain the correct answer using the SOR solver, it is necessary to have an overall electroneutral system. Ludwig already contains a test for the SOR routine. Thus, it would be simple, while also accurate, to compare the FFT solver to the SOR solver using the already existing test case. The current test case sets a uniform wall of charges at the $z = 0$ and $z = L_z$ boundaries of the system and fills the inside uniformly such that the system is overall charge neutral. The SOR solution is then compared to that

of the Gauss-Jordan method. In order to test the FFT solver it will be sufficient to test it against the SOR solver for this problem. When comparing the solutions, one must remember that the final answer may be offset by some constant value. This problem does not have an exact analytical solution so it will serve only as a guide to whether or not a consistent answer is being obtained with the FFT solver.

2.5.1 Gouy Chapman Problem

The Gouy-Chapman problem deals with a flat surface with a specified surface charge. The electrolyte is symmetric and counter-ions are placed as necessary for the SOR solver to function correctly. Thus the system is overall electroneutral and it models an electric double layer. There is an analytical solution to this problem in the one-dimensional case in the low potential approximation and the results from running Ludwig can be compared with this solution. The expected solution is of the form

$$\Psi(x) = \Psi_D exp(-\kappa x)$$

where $\Psi(x)$ is the potential and Ψ_D is the Stern potential at the surface of the wall and is realted to the surface charge. κ is the inverse Debye length, $\kappa = l_D^{-1} = \sqrt{8\pi l_B I}$. $l_B = \frac{\beta e^2}{4\pi\epsilon}$ is the Bjerrum length with $\beta^-1 = k_B T$, e is the unit charge and $\epsilon = \epsilon_0\epsilon_r$ is the dielectric permittivity. Finally I is the ionic strength of the electrolyte given by $I = \frac{1}{2}\sum_k z_k^2 \rho_{B,k}$ with z_k as valencies of species k. $\rho_{B,k}$ is the bulk charge density of species k not at the wall.

As can be seen in figure 2.6 the results were close to the theory. The documentation for Ludwig contains the parameters that were used in the original tests are given and these can be used to repeat the results. They are also documented in the appendix, section C.2. Reproducing the graphs generated with the SOR solver, by using the FFT solver would give a positive indication that the FFT solver is working correctly.

2.5.2 Liquid Junction Problem

Timing of the FFT solver was conducted on one other real world problem, where the problems with the scalability of the SOR solver were first seen. In the liquid junction problem two liquids, each containing electrolytes of slightly different concentration and diffusivity of the charged species, are brought into contact. The charges move such as to equalise the charge, so charges from the higher concentration regions diffuse to regions of lower concentration. Since the two species have different diffusivities however, they migrate at different speeds and this causes parts of the system to remain charged. For the full description of the effect refer to [26]. Again, the input parameters are given in the documentation and the appendix. These can be used to verify that the solution is correct before timings are conducted.

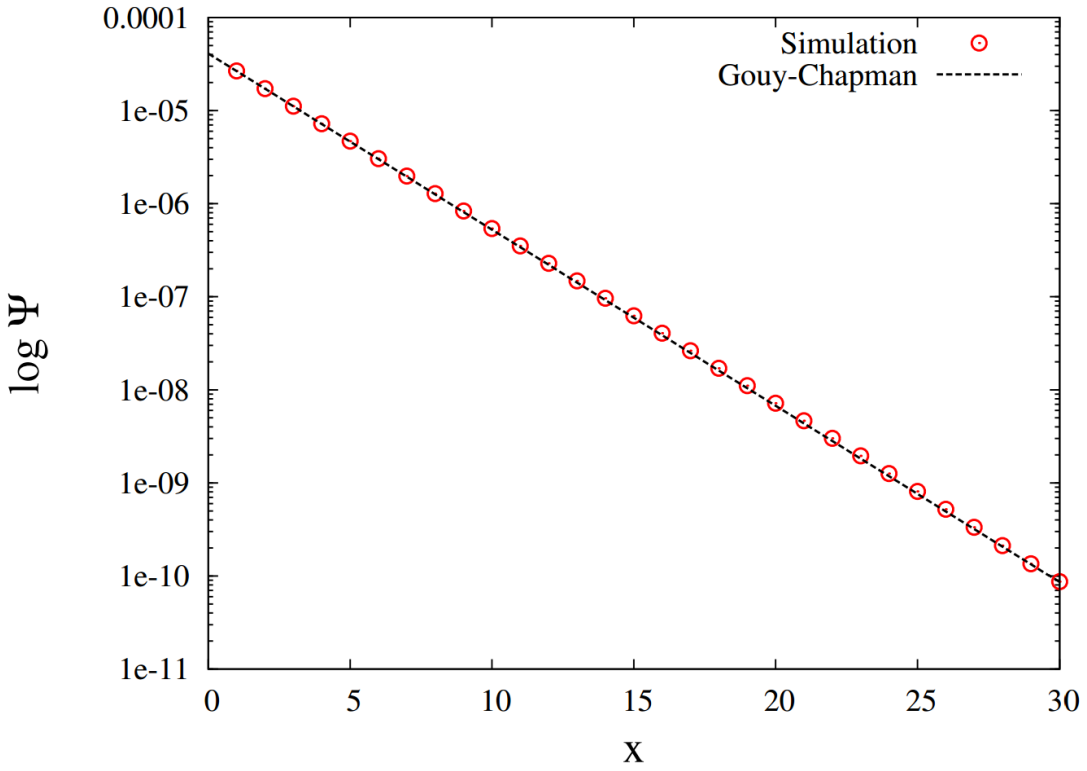


Figure 2.6: Plot of Ψ against position for the Gouy-Chapman problem comparing the analytic and simulation results with $\rho_{0,\pm} = 1 \cdot 10^{-2}$. Reproduced from [24].

2.6 High Performance Computing and Performance Metrics

The problems that are dealt with in Ludwig are in general quite large. This means that, while the program has been written with the hope that laptop and desktop machines can run it, usually high performance computing resources are necessary to compute the solutions for large problems. For this project the two main systems that will be used are HECToR[17], the UKs national supercomputing service and Bluegene/Q[3], an IBM machine run by EPCC in the University of Edinburgh. A more thorough discussion of the specifics of these machines will be presented later.

When building parallel applications, the main objective is that they will perform well on a large number of processing elements. Measuring how well a program performs from this perspective is usually done with the use of a metric called the speedup. Ideally if n processors are used on a problem, the solution should be found n times faster. The actual speedup of a piece of code can be found from:

$$Speedup(N) = \frac{T_{serial}}{T_N}.$$

Here, T_{serial} is the time taken to execute in serial, and T_N is the time taken to execute on N processors. With effective parallelisation the value for speedup should be close to the

number of processors used. There are external factors that can cause this relationship to deviate from the expected linear behaviour. For example, when using more processors there may be more cache available, resulting in a speedup of the code. On the other hand an increase in the number of processors will inherently increase the amount of communications and will impact negatively on the performance. Modern supercomputers usually have a mixed architecture, with shared memory nodes connected together on a network. When running jobs on these machines, it is usually necessary to request full nodes. This means that if the number of cores used is less than the node size, the cores have access to more memory and cache than they would otherwise. As a result of this it is common to compute speedup relative to the number of cores in a node and this is what will be done in this project.

Normally this metric is used with a fixed problem size. This case is known as strong scaling. The opposite of this is weak scaling, where the problem size is increased as the processor count is increased. More precisely, the amount of data that must be processed by each compute core remains the same. The latter is easier to achieve, but this project will look at both.

Another term that will be used is the bi-sectional bandwidth. The bandwidth available on a network describes the rate at which data can be transferred through it. In order to compute the bi-sectional bandwidth of a parallel machine one can split the processors into two equally sized groups. These groups should be chosen to have the smallest number of network connections possible between the two groups. The bandwidth across the split is the bi-sectional bandwidth.

2.7 Programming

Ludwig is currently written in ANSI C (1989) with the exception of calls to the library function `erfc()` in the Ewald summation code. Still, most of the code attempts to use abstract data types, which are usually more suited to object-oriented languages. To avoid the overuse of `void` pointers that would arise from the use of C and abstract data types, the criteria for encapsulation have been relaxed in some parts. For the parallelisation of the code, the message passing library MPI has been used. This allows for much larger numbers of processors to be used than shared memory programming, which is limited to not many more than 100. Ludwig has been successfully run on up to 131,072 MPI tasks[24]. The code has been written with the intention that it can be run in serial, through the use of an MPI stub library and a parallel environment.

Chapter 3

Design & Implementation

In this section we shall discuss the main design of the code written, and some of the specifics of how it was written. We will also mention the target architectures and how the problems were actually run on these machines. Finally a discussion of the accuracy tests and timing tests will be presented.

3.1 Target Architecture

Two main systems were used for this project, HECToR and BlueGene/Q.

HECToR is the UK's national supercomputer service, though it will be replaced by ARCHER at the end of 2013[17]. HECToR is a Cray XE6 system with that uses 16 core AMD Opteron 2.3GHz processors. Each node contains two of these processors and is coupled with a custom built Cray Gemini interconnect to connect the processors in a 3D-torus. These links have a peak bi-directional bandwidth of 8GB/s with latency of 1-1.5μs. This allows for a high MPI point-to-point bandwidth and a low latency[17]. There is 32GB of memory available per node which is shared between all the cores in a node. Altogether, 2816 XE6 compute nodes are offered, resulting in a total of 90,112 cores with a theoretical peak performance of 800Tflops. On HECToR speedup will be referenced to 32 cores as this is the node size.

Bluegene/Q is hosted by EPCC as part of the DIRAC project, the UK's supercomputing facility for theoretical modelling. It is composed of custom chips with 16 1.6GHz Powerpc64 A2 cores, each of which is capable of 4-way multi-threading. One of these 16 core processors comprises a node and in total there are 6144 nodes, meaning a total of 98,304 cores. This can show a peak performance of 1.26Pflops. The memory available is 16GB per node, again shared between all the cores in the node. The interconnect is also custom built and is a 5D-torus, capable of 40GB/s[3], allowing for extremely high quality communications. For measurements, we will reference speedup to the time taken using 16 cores on BlueGene/Q.

It is believed that both systems should perform well when conducting FFTs. Due to the slower clock speed of the processors, BlueGene/Q is expected to perform worse than HECToR on low core counts. However there is an excellent interconnect and as a result the scalability should be better than that on HECToR.

For development and debugging purposes MORAR was used. The architecture of this machine is similar to HECToR, but with one AMD Opteron processor comprising a node. It also uses a factory standard interconnect. This is a machine maintained by EPCC primarily for the use of MSc students in the University of Edinburgh. Timings were not conducted on this machine as there is a maximum of 64 cores available. This is too small to provide any interesting results but since it is less busy than the aforementioned machines, queue times are much shorter.

3.2 Three dimensional FFT Libraries

It was decided that due to the amount of time available for this project, a library would be used for computing the 3D FFTs. Implementing an efficient 3D Fast Fourier Transform algorithm would be an entire project in itself. There are two main 3D FFT libraries, P3DFFT[29] and 2DECOMP&FFT[23], both of which show similar performance[5]. P3DFFT is probably the most well known and it is a dedicated FFT package whereas 2DECOMP&FFT has extra functionality such as routines for allocating arrays that were not needed for this project[23]. Although both libraries are written in Fortran, P3DFFT provides a C interface. This is not the case with 2DECOMP&FFT. For these reasons P3DFFT was the library of choice. P3DFFT takes care of all of the messages necessary to perform 3D FFTs and provides routines to determine the sizes of the local arrays on each processor. It then uses the FFTW[12] or IBM's ESSL[4] library to perform the actual transforms in each dimension.

For this project version 2.5.1 of P3DFFT was used. On HECToR it was built using FFTW 3.3.0.1 as the base, which was already installed. On the BlueGene/Q machine it was built on ESSL 5.1.1-0.ppc64, again already installed. For development purposes the School of Physics computer lab machines and MORAR were used and here FFTW version 3.3.3 was used. FFTW and P3DFFT are freely available according to the GNU General Public License as published by the Free Software Foundation, however ESSL is an IBM proprietary library and must be purchased.

3.2.1 Installing P3DFFT

P3DFFT has been written with four main compilers in mind. These are the GNU, PGI, Intel and IBM compilers. It provides wrapper functions to make calling from C easier. This means that when compiling to use with C, certain linker libraries need to be used to ensure Fortran intrinsic functions are recognised. Due to its nature as a Fortran module, the names of functions in the object file may also differ depending on the compiler.

Currently this is resolved by using compiler-specific `FORT_MOD_NAME` macros but as a result compiling with any other compilers is extremely difficult. On HECToR the PGI compiler was chosen as the best option. This is known to produce faster code than the GNU compiler. The Cray compiler that is also available was not deemed to be an option, due to the complication of compiling with it.

P3DFFT provides a configure script for generating makefiles. Using this it was simple to install on HECToR. The exact command used was

```
./configure --prefix=/home/d45/d45/s1256564/work/build/
p3dfft-2.5 --enable-pgi --enable-stride1 --enable-fftw
--with-fftw=/opt/fftw/3.3.0.1/interlagos FC=ftn CC=cc
FCFLAGS=-O3 CFLAGS=-O3
```

Note the use of the `-O3` flag here to ensure optimised code was produced. On HECToR, tests were conducted with and without the `-O3` flag specified. The `test_sine` program provided as part of P3DFFT was used as this prints the amount of time taken per FFT forward and backward solve in seconds. When using 1 MPI task, time per loop averaged over 25 iterations, the time per iteration was 0.082s with the `-O3` flag, whereas without it the time taken was 0.095s. Thus, it was decided that this flag should be used for all timing .

On BlueGene/Q installation was somewhat more complicated. A number of issues were encountered with both the configure script and the code itself when being used with the ESSL library. As a result some work was necessary before the library could be successfully used.

The first problem was that in order to compile MPI code with the IBM compiler, the compiler wrappers `mpixlc_r` and `mpixlf90_r` were used. These did not, by default, know the path of the ESSL library. In general IBM compilers can find the library simply by passing the `-lessl` flag, and this was the case for the serial compilers `xlc_r` and `xlf90_r`. The configure script of P3DFFT assumes that passing the `-lessl` flag will find the library, so it was necessary to change the configure script slightly. The proposed solution was to edit the configure script to stop it assigning the `-lessl` flag in the Makefile and the path to the library was included manually in the Makefile.am files in the appropriate locations. The configure script was then run with the following command:

```
./configure --prefix=/home/e01/e01/rshort/p3dfft/
--enable-ibm --enable-essl --enable-stride1 CC=mpixlc_r
FC=mpixlf90_r CFLAGS=-O3 FCFLAGS=-O3
```

Again the `-O3` flag was used for the final tests that were run.

Once this problem was resolved the code was successfully built and installed. The library has testing routines as part of the source code and these were executed. On HECToR the correct answers were obtained but it was found there was a problem with allocating some arrays on BlueGene/Q. The correct answer would be obtained but the program would crash. Again, this was a problem associated with the ESSL library but

this time to do with the actual P3DFFT code. A number of `#ifdef` statements are included in the code, as the different base FFT libraries require different input. When the ESSL library was being used the code attempted to allocate some arrays twice. They are also deallocated twice, which caused the crash. It was again possible to fix this with some relatively small changes to the code. Simply adding a test (using the `allocated()` function available in Fortran) to see if the arrays were allocated before allocating or deallocating. A final problem was discovered when trying to run the code on one MPI task on BlueGene. This was to do with the interaction between the P3DFFT library and the ESSL FFT routines. Unfortunately the cause was not found in this case. As the main goal of this project was a parallel FFT library implementation it was not deemed of too much concern. As will be discussed later, it is intended to replace the library with code written by the authors of Ludwig to improve the ease of use.

A `.tar` file was created containing the changed version of the offending files and this is available from the author upon request. The exact changes that were required are given in appendix.

3.3 Integrating with Ludwig

It was necessary to make some changes in the Makefile used in Ludwig in order to use the P3DFFT library in combination with it. The fact that P3DFFT is a Fortran library means that it requires some extra flags to allow the C compiler use some of the built-in Fortran routines. On HECToR the addition of the `-pgf90libs` flag, along with links to the P3DFFT and FFTW library locations were all that was necessary. Bluegene/Q needed more flags, as there is no single flag that provides all of the linking libraries with the IBM compiler. Also passing of the locations of the libraries was required. The complete list necessary for successful compilation is included in Appendix A.1.

In order to choose between using the SOR and FFT routines, function pointers were used. The appropriate function is then pointed to at initialisation. The user can change which solver to use by changing a variable in the input file at runtime. A full list of the changed files is available in the appendix.

3.4 Cartesian and Pencil Decompositions

In order to use FFTs to solve Poisson's equation in Ludwig it was required to write code that would first execute the decomposition swapping before computing the solution of the equation. Ludwig always uses a three dimensional Cartesian processor decomposition. As mentioned earlier, in order to avail of a large number of processors when computing FFTs it is necessary to use a pencil decomposition. As such it was necessary to write code that would transform from the Cartesian to the pencil decomposition and back again.

It was decided that this would be done via four main routines:

- Initialisation.
- Cartesian to pencil.
- Pencil to Cartesian.
- Clean up.

This was chosen as it was believed that the set up required for initiating the transformation should only be done once. The actual solution of Poisson's equation will be needed thousands of times in a typical run of Ludwig, thus the amount of work to be done when computing this should be minimised. To achieve this, variables can be created and given values in the initialisation phase. These variables are declared in file scope with the `static` keyword to prevent external routines accessing them, while allowing internal functions access to the values later in the execution. This allows for data structures to be created in the initialisation and then used when actually conducting the decomposition switching. As a result a clean-up routine is also necessary to ensure all memory is freed and default values are reset.

The method chosen of communicating the data between processes was point-to-point communication using `MPI_Isend`. Other methods such as defining sub communicators and using Alltoall transformations were looked at but deemed too complicated to implement easily. Thus, initialisation must include determining the corresponding processors for swaps in both directions and setting up a method to execute these. An array of `MPI_Subarrays` is used to keep a reference of which data corresponds to which destination processor. This is much simpler to use than `MPI_Vectors` as `MPI_Subarrays` are designed for use in multidimensional arrays and are associated with a specific part of the array. Only the sub-array needs to be stored, and not the starting address of the part of the array it corresponds to. These can then be used in all subsequent communications. The other information that must be computed is the number of processors that must be communicated with. Due to it being necessary to allocate the various arrays, this is an important quantity.

Finally, it was decided to use this module to initialise the P3DFFT library. Before it can be used to compute Fourier transforms, the dimensions of the problem must be passed to P3DFFT. It also takes a variable that allows for overwriting of the input transform, meaning a saving on memory, which will be valuable for larger problem sizes. P3DFFT then provides a routine to allow the user find the sizes of the local pencil arrays, in both real and Fourier space. This routine also returns the global starting address of the arrays, which is useful for computing the k^2 value. In order to reduce the number of calls to the library from outside the decomposition switching module, the arrays of sizes and starts were declared with the `static` keyword and routines written to provide access to these arrays. To further save on memory, it was decided to use the same array in the pencil decomposition for both spaces. Fourier space is made up of complex numbers, so twice the memory would be needed to store the data points. However, it is also periodic over the interval and half of the data in one dimension does not need to be stored. This

means that it is not immediately obvious how much memory should be allocated for storing the pencil arrays. A routine which would return the larger of the two spaces was thus written to allow for easy allocation.

3.4.1 Decomposition Switching Implementation

P3DFFT requires the processor dimension as input so it is necessary for either the user to give this or it to be calculated by the program. The Ludwig runtime environment allows for the user to set values in an input file at run time. An extra option was added to this input file to allow for the pencil decomposition grid size to be specified. It was also necessary to provide a default pencil decomposition for when the user does not specify one, or an invalid one is specified. The MPI function `MPI_dims_create` is useful in this instance. In general the best performance is seen when this grid is as close to square as possible[5] [29]. This is usually the output of the `MPI_dims_create` function, although it can be implementation dependent.

In the case where the Cartesian grid is also two dimensional, it was possible to simply copy the data between two arrays on the same process. The same array cannot be used for the entire program as the Ludwig arrays contain halo data and this must be removed before passing it to P3DFFT. It is important to note that the pencil array must be exactly the same, apart from these halos, as the two dimensional Cartesian array. If they differ in any dimension the Fourier transforms will be incorrect. Thus some fairly rigorous checking is required upon initialisation. This only needs to be done once as a variable can be set to ensure the correct transformation operation between decompositions is performed.

For all other cases, significantly more work is necessary. We will refer to figure 2.5 to aid in the understanding of this section. From this figure, the x direction is referred to as the column, y as row and z as depth. We also note that z is the direction of contiguous memory. The first step is to compute how many messages must be sent. This can be done by looking at the destination process of the first point in the local array, and then iterating over the array until each element has been associated with a destination. The size of the sub-array at each point can be used to skip over other points that have the same destination process. This means that only a subset of the local array elements need to be iterated over. Interestingly, the amount of processing necessary actually increases with larger processor counts as more messages will be sent. Following from this, the arrays for the sub-arrays and list of destination processors can be allocated.

Now, we have somewhere to store the data so the computation proceeds to find which processes must swap with which using the formula in equation 2.16. We will look in detail finding the destination from the Cartesian grid. The computation for the processes to receive from in the pencil grid proceeds separately and is similar. Alternatively, it would be possible to compute the destination processes in the Cartesian decomposition and then send the information. The receiving processes would then post wildcard receives. The problem here is that it is difficult to work out before hand how many

receives are necessary, without doing most of the work towards finding out what processors to receive from. It would also be unnecessary communication and for this reason the former method was used.

To save on computation, the current implementation computes the sizes of the sub-arrays to be sent (and consequently received) and sends these sizes to the receiving process. It is important to remember that the Cartesian arrays in Ludwig have halos, though the pencil arrays do not. This is done after the computation of the number of destination processes, though there is some overlap in the computation. The sub-array sizes mostly correspond to the size of the destination array, but care must be taken to ensure the point corresponds to the first one in the destination array. This can be done by subtracting `global_coord%pencil_size` from the sub-array size. That sub-array can then be skipped over to the start of the next one. This is done in the rows, and then columns. For the computation of the destination processes in the Cartesian decomposition, note that only the row and column information is necessary. The processes in the pencil decomposition have one entire dimension stored on them, the dimension that is contiguous memory. Thus for any point with a given row and column coordinate the destination process is the same regardless of the depth coordinate.

The receiving processes conduct a similar set of operations but they do not need to compute the sub-array sizes, as these are received, and equation 2.17 is used. Also, it is much easier to guess at the number of processes that will need to be received from and accurately allocate memory based on this. Each process must receive one message from each processor in the z-direction. If, after these have been accounted for, there is more data to be received, at least twice as many processes must be received from i.e. another full depth. If there is still more data to be processed, the total messages must be at least twice this again, two further full depths. Both the x and y sizes of the pencil arrays are in general the same as, or smaller, than the x and y sizes of the Cartesian arrays. Thus, having to send four full depths is unusual. It is possible for the user to specify pencil decompositions where this is not the case, though these will show poor performance for the FFT as well as in this part of the code. For these reasons the arrays were allocated with the minimum number of elements and then reallocated to be twice as large if the arrays became full. For transforming back from the pencil decomposition, the same sub-arrays and destination processors can be used as it is just the inverse transformation.

Actually swapping between the two decompositions is then simply copying between arrays, or a number of `MPI_Isends`. Non-blocking communication is used throughout to allow for calculation to proceed simultaneously to communication.

3.5 FFT routine

In order to solve Poisson's equation another routine was written for tha main computation. The first step is to find σ, the Fourier transform of ρ, the electric charge. Ludwig

has the capability of modelling different charge carriers. These are stored in one large array with multiple entries for each lattice point. In order to compute the actual charge density a routine is provided, `psi_rho_elec`, that returns the total charge at each point. This is used to fill an array with the charge values.

This array is reordered so the data is in the pencil decomposition the Fourier transform can applied to it.

Now equation 2.9 can be solved. It is important to note that the library being used computes a non-normalized transform. Thus computing the forward and backward transforms will multiply the input by N, where N is the total number of points in the system, and this must be accounted for. In order to save on the number of divides that would occur in the routine, the entire divisor was computed before the divide. This means that σ was multiplied by a factor of $1/(N\epsilon k^2)$. The reason for this is that divides are not hardware pipelined and thus impact severely on performance.

Following this the inverse Fourier transform can be computed with the answer being the desired electric potential, Ψ.

The last step is to transform the data back to the Cartesian processor decomposition so that Ludwig can continue with the rest of its operations.

We will now discuss in more detail some of the more complicated parts of this routine.

3.5.1 k^2 multiplication

When computing discrete Fourier transforms, there are different conventions for numbering the frequencies in Fourier space. It is important to follow the conventions of the library or method being used. P3DFFT uses FFTW to do the Fourier transforms so the FFTW documentation is relevant in this case. It was noted that the output was in the standard order, that is "the k-th output corresponds to the frequency k/n"[13]. ESSL also uses this convention, improving the portability of the code. A clearer way to state this is in terms of frequencies. The positive frequencies are stored in the first half, while the negative frequencies are stored in backwards order in the second half of the output. The simplest way to achieve this is to compute an adjusted lattice point index, l, with $-N_x/2 \leq l < N_x/2$ and use this in equation 2.15 to find the k_x values. Given is the algorithm used to compute the k_x values at each point, using `ix` to represent l:

```
ix = global_coord[X] - n_total[X]*
                    *((2*global_coord[X])/n_total[X]);
kx = (2.0*pi/n_total[X])*iz;
```

Note that integer division is exploited here to determine whether the global coordinate is in the first or second half of the interval when computing `(2*global_coord[X])/n_total[X]`. This gives values $-\pi \leq k_x < \pi$ with positive frequencies in the first half of the interval and negative frequencies in the second half, as required by the convention.

The P3DFFT library provides useful calls to give the global position of the first element of the local array, allowing for simple computation of the global coordinate values. They must be manually updated as the array is iterated over.

k^2 is then computed using equation 2.14. The actual code used is:

```
k_square = 1/((kx*kx + ky*ky + kz*kz)*epsilon*
                *(n_total[0]*n_total[1]*n_total[2]));
```

In the case $kx = ky = kz = 0$, k^2 is set to be zero. Then `k_square` is multiplied by the Fourier transformed ρ array.

There are two other important considerations in this part of the code. Firstly, as P3DFFT is a Fortran library, it considers the arrays to be contiguous in memory in the first index. C uses the opposite convention, memory contiguous in the last index. This is not a problem when indexing arrays returned from P3DFFT, as they can be addressed in the usual manner in C. An issue does arise when considering arrays that contain the information about the sizes or positions of other arrays. For example, the values of the `istart` array used in the code were created by P3DFFT. It contains the global address of the first element of the local Fourier transform array. The value of `istart[0]` is information pertaining to the contiguous memory direction (z-direction). Ordinarily, when using C, this information would have been stored in `istart[2]` (for a 3D problem). Care must be taken to ensure the correct elements of the array are used.

To further complicate matters, after computing the Fourier transform, P3DFFT leaves the data in transposed form. The transposed decomposition can be seen in figure 3.1.

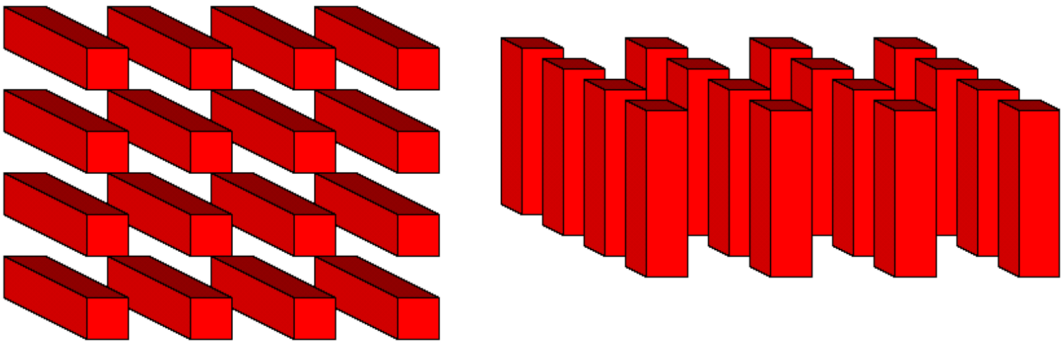


Figure 3.1: Diagram showing how the data is decomposed across the processors before the FFT (left) and after the FFT (right). The decomposition stays the same regardless of the `stride1` flag. It only changes the local memory arrangement. Reproduced from [18]

When building the library, the user has the option to specify a flag, `-enable-stride1`, recommended by the author as it can show a significant improvement in performance[29]. It was thus used for all timing runs of the code. This flag changes the ordering of the transposed array in memory. When not specified, the array retains the original ordering from before the FFT was conducted. If specified, the array changes to the opposite

ordering after the FFT, i.e. the direction that was contiguous before the transform is now the least contiguous. The result of this is a change in the lattice point that each element of the array corresponds to depending on whether the flag is specified or not. The P3DFFT library defines a preprocessor macro if the flag was used. Thus an `#ifdef` preprocessor directive was used to check at compile time if the library has been built with the `stride1` setting enabled. This directive is defined in the file `config.h` which is not automatically copied to the include directory. It must be copied manually to ensure the correct calculation is performed. Depending on the result, the order in which the directions are iterated over is switched to ensure the next lattice site corresponds to the next one in memory. Care must also be taken to ensure the sizes of the local Fourier transformed array correspond to the correct physical direction in the simulation, as these are opposites in the two cases.

3.6 Testing

3.6.1 Unit tests

A unit test was written for the decomposition switching code. This test initialises each location on the lattice with a unique number based on its location in the global grid while in the Cartesian decomposition. Then the switching takes places and each lattice point is tested to check if it has the correct value in the global grid while now in the pencil decomposition. The decomposition can then be switched back and the values can be checked again.

A unit test already exists in Ludwig for the `psi_sor_poisson` update routine. This test consists of a system with two uniformly charged walls at $z = 1$ and $z = L_z$. The rest of the system is initialised with charges to ensure the overall system is charge neutral. Comparison with a Gauss Jordan routine is how the accuracy is then verified. In order to test the `psi_fft_poisson` routine, the initialisation of this test was used, and the results of the FFT were compared with the SOR results. This was written as a new unit test however, to ensure that each file has a separate test.

3.6.2 Timing

Ludwig provides timing routines and these were availed of. These actual times are taken using the `MPI_Wtime` function, which returns the wall time. Using these timers was simpler than writing new ones. A routine, `TIMER_statistics`, is provided that prints the values of all the timers that have been used in the code. The values printed are the minimum time, maximum time, average time and number of calls. It is important to ensure the number of calls is large enough to give an appropriate average time for the computation. This allows for easy collation of the data at the end of a timing run, especially if multiple parts of the code are being tested. The code in Ludwig makes

heavy use of assertions to ensure accuracy. These were turned off for timing runs to improve speed and the flag `-O3` was passed to the compiler to allow it to perform optimisations.

3.6.3 Gouy Chapman Reference Problem

The Gouy Chapman problem is a 1 dimensional problem. However, using a one dimensional grid limits the number of processors that can be used when solving with FFTs to one. For this reason the grid used was $64 \times 8 \times 8$. The data was then replicated, such that the system was composed of many one dimensional problems and thus the testing will actually test the 3 dimensional nature of the FFTs, while still appearing one dimensional. Tests were conducted using 8 and 16 cores in order to test the various decomposition possibilities.

In order to verify the accuracy of the FFT solver plots of the electric field in the system, Ψ, were created after a large number of steps. These plots should agree with the one presented in figure 2.6

3.6.4 Liquid Junction Potential Problem

The liquid junction problem is a truly three dimensional one. As well accuracy testing, timing tests were conducted on this problem. Accuracy tests were performed on problem sizes of 128^3 and 256^3. The problem sizes used for timings were 64^3, 128^3, 256^3, 512^3 and 1024^3. It should be noted that the number of points in the 256^3 system is $1.6 \cdot 10^7$. This is very large, and storing a double for each one of these points can pose memory problems. On BlueGene/Q the 256^3 problem could not be run on less than 8 cores and the 512^3 problem not on less than 128. On HECToR, where there is twice as much memory available per node the 256^3 problem could be successfully executed on one node. The 512^3 problem must use at least two nodes however, so 64 was the minimum number of cores used for this. It is also worth noting that running the problem with this problem size can produce output files totalling up to 26GB. The user must be careful that the required space is available. Problem sizes of 512^3 and larger would not in general be used for physical problems for this reason, but for the purposes of this project they were investigated in order to better understand the performance of the solver.

When it came to timing tests, core counts ranged from 2 to $32,768$. Most tests were conducted with the nodes fully subscribed, though some were done with under-subscribed nodes in order to determine if a performance benefit could be seen. The data obtained from these tests could then be applied to understanding both the strong and weak scaling of the code along with looking at the absolute time taken.

At this stage some timings were also taken for the decomposition switching initialisation, to determine if it was time critical and needed more work or not.

3.6.5 Scripts

Further to this a bash script was written to combine the data into easily readable formats for plotting. This also allowed for consistency when extracting the results for a particular run of the code.

Chapter 4

Results

The results have been divided into two main sections. First the results of checking the accuracy of the FFT solver are presented and discussed. Following this, the main timing results will be shown and further analysed. Note that all physically measured quantities will be presented in terms of simulation units.

4.1 Accuracy

In this section the results of the accuracy testing will be presented. Tests were conducted on both BlueGene/Q and HECToR to allow for any errors that may be caused by the different base libraries. This also tests that the implementation itself is hardware independent which is important in Ludwig.

4.1.1 Single Solve - Unit Tests

First the decomposition switching was tested. This test was conducted on various core counts and problem sizes to ensure all possibilities were accounted for. When it was found to be working correctly, development and further testing of the FFT routine could continue.

Next, testing of the FFT routine was conducted on the unit test outlined in section 2.5. Comparing the FFT solution against the SOR solution showed that the general form of the results on the 64^3 grid size for this problem are the same. Figure 4.1 shows a plot of the potential in this system, with the x and y coordinates held constant and the z coordinate varied. The x and y coordinates are not as interesting as the potential is constant across them.

From this graph we see that the shape of the potential is consistent. However, if we look in more detail at this plot, the values at the edges of the system are not as similar as were originally expected. In this example, the value from the FFT solver was 0.001271

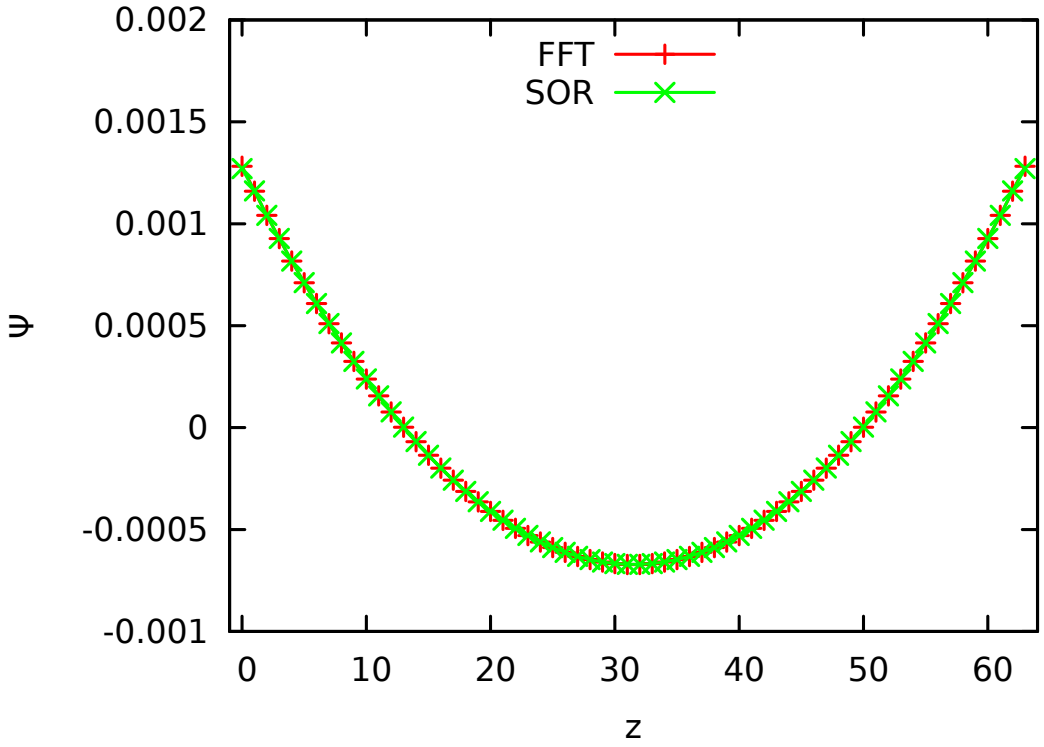


Figure 4.1: Plot of the potential with varying z coordinate in a 64^3 system for the unit test. The shape generated by the SOR and FFT solvers is the same.

on the boundary points, while the SOR solver produced a value of 0.001282. This is a fractional difference of about 0.9%. Usually two different algorithms of the same method could be expected to have a difference of less than 0.01%, with differences only being caused by numerical inaccuracies.

In order to understand this further the fractional difference was plotted against the position. This should show where the difference comes from. As can be seen in figure 4.2, this fractional difference varies depending on the system size, with smaller grids giving a larger error. It also shows that the main source of error is the boundaries. The error spikes near 20 and 100 on the z axis are due to the fact that the potential is close to zero at these points. As a result, computing the fractional difference entails dividing by a small number, so the results are not informative. In figure 4.2 some of these points have been removed to improve clarity. A complete plot can be seen in the appendix in figure C.1.

The variance with system size can be explained by poor discretisation. The FFT algorithm multiplies by $1/k^2$ in Fourier space. Smaller grids approximate the space less well than larger ones. This could result in incorrect values being found on the smaller grids.

It is important to note that, in reality, the SOR and the FFT solvers are different methods of solving Poisson's equation. With the SOR method, the derivative is approximated by

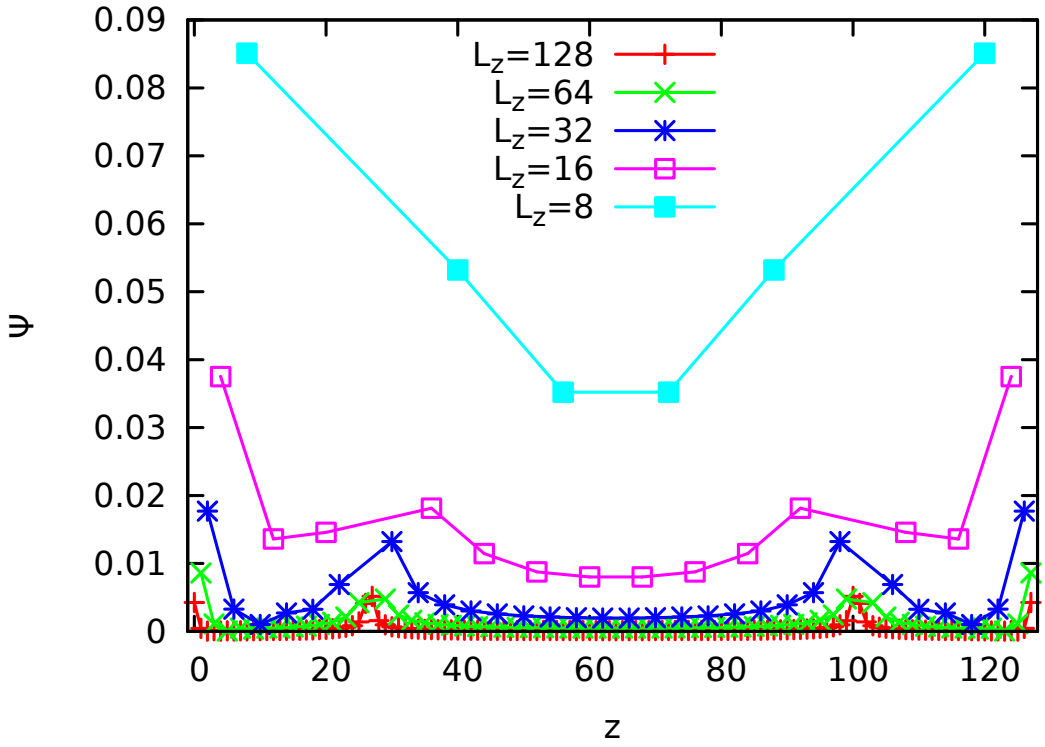


Figure 4.2: Plot of the fractional difference between the SOR solver and the FFT solver for the unit test problem sizes from $L_z = 8$ to $L_z = 128$. L_x and L_y were held constant at 64. The z positions have been scaled so all data sets will fit onto the same plot. This shows how the error becomes more prevalent with smaller sizes. Some points have been removed where the fractional error is distorted due to dividing by a small number. A plot with these points included can be seen in the appendix in figure C.1.

a finite difference and iteratively solved. The FFT solver approximates the derivative in a different way, multiplying by ik in Fourier space. This is where the deviations between the two methods stem from. For the particular problem being solved in the unit test, there is no analytical solution. There is only an approximate one. The fact the the SOR and FFT solvers show similar behaviour shows they both approximate the solution well. However, it is not possible to say which is exactly correct. Still, the total potential in the system is the same to a high degree of accuracy, with a difference of approximately $1 \cdot 10^{-13}$ when the total potential is approximately $1 \cdot 10^{-3}$. The FFT solver was also noted to have an advantage over the SOR solver. The FFT solver does not require the system to be overall charge neutral, which will allow for new problems to be simulated in the future.

In general, $L_x = 64$ is the smallest length that is used when looking for interesting physical results. Larger grid sizes are used more often. This means that the differences displayed on small grids are not a problem for general cubic problems, and become less so the larger the problem.

If long, thin systems are to be modelled there will be some issues. In this case the poor discretisation in the shorter dimensions will reduce the accuracy of the solution. To determine the effect of this problem different grid shapes and sizes were experimented with. It was found that systems with high aspect ratios (e.g. $256 \times 4 \times 4$) will not obtain the correct results. In general, the smallest the grid should be in any dimension to obtain correct physical simulations is 32, though larger is better if possible.

It is difficult to quantify exactly how problematic this error is. It changes depending on each grid dimension. Below 32 in any dimension, it cannot even be guaranteed that a reasonable solution will be found. For example, using a $256 \times 4 \times 4$ grid will produce results that differ from the SOR solver by up to a factor of 10. The shape of the potential generated is not even similar. When used in the main part of Ludwig, 'not a number' errors appear due to the large inaccuracies in the solution to Poisson's equation. On the other hand, solving a $64 \times 4 \times 4$ grid with the FFT solver gives only slightly different results from the same solver on a 64^3 grid.

This has an impact on comparing the performance of the SOR solver and the FFT solver. When using the SOR solver for physical modelling, grid sizes such as $256 \times 4 \times 4$ would be used, with the system periodic in the second and third dimensions. This leads to the effect of modelling a larger system but using a smaller number of actual points. The FFT solver would need a problem size of at least $256 \times 32 \times 32$ in order to reliably obtain the correct answers. As a result of having less lattice points it is possible the SOR solver could produce a result for this problem in a shorter time. On truly three dimensional problems it is still expected that the FFT solver will perform better. This was then introduced as a test to be conducted in the liquid junction potential problem.

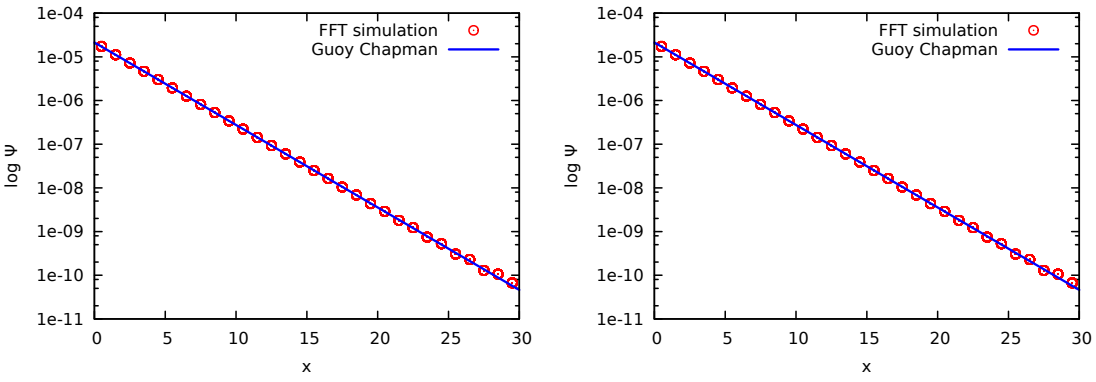
The unit test was also used to test the algorithm in the case that the `-enable-stride1` flag had not been passed to the P3DFFT library at build time. In this case, the same results were found as when the flag was passed.

At this stage, it was not known how much of an impact the small differences near the boundaries would affect physical simulation. Thus it was decided to continue and determine if the physics modelled were correct.

4.1.2 Gouy Chapman problem

Testing was conducted on the Gouy Chapman problem on a $64 \times 8 \times 8$ grid as originally intended. The results from the Gouy Chapman problem are presented in the form of plots of the potential compared with the analytical value in figure 4.3.

This problem was initialised with zero potential and $\rho_{0,\pm} = 1 \cdot 10^{-2}$. Comparing these plots from both Bluegene/Q and HECToR with figure 2.6, we see the same behaviour as has been modelled before. The parameters used were the same as those used in the original test and these are given in the appendix C.2. After the simulation is run, the potential has a negative value in the centre. In the SOR case this was $\Psi_c = -2.364 \cdot 10^{-6}$, whereas the FFT solver produced a value of $\Psi_c = -2.3307 \cdot 10^{-6}$. The origin



(a) HECToR *(b)* BlueGene/Q

Figure 4.3: Results for the Gouy Chapman problem on HECToR and BlueGene/Q. The plots compare the simulation results when using the FFT solver with the analytical solution for $\rho_{0,\pm} = 1 \cdot 10^{-2}$. This can be compared against figure 2.6.

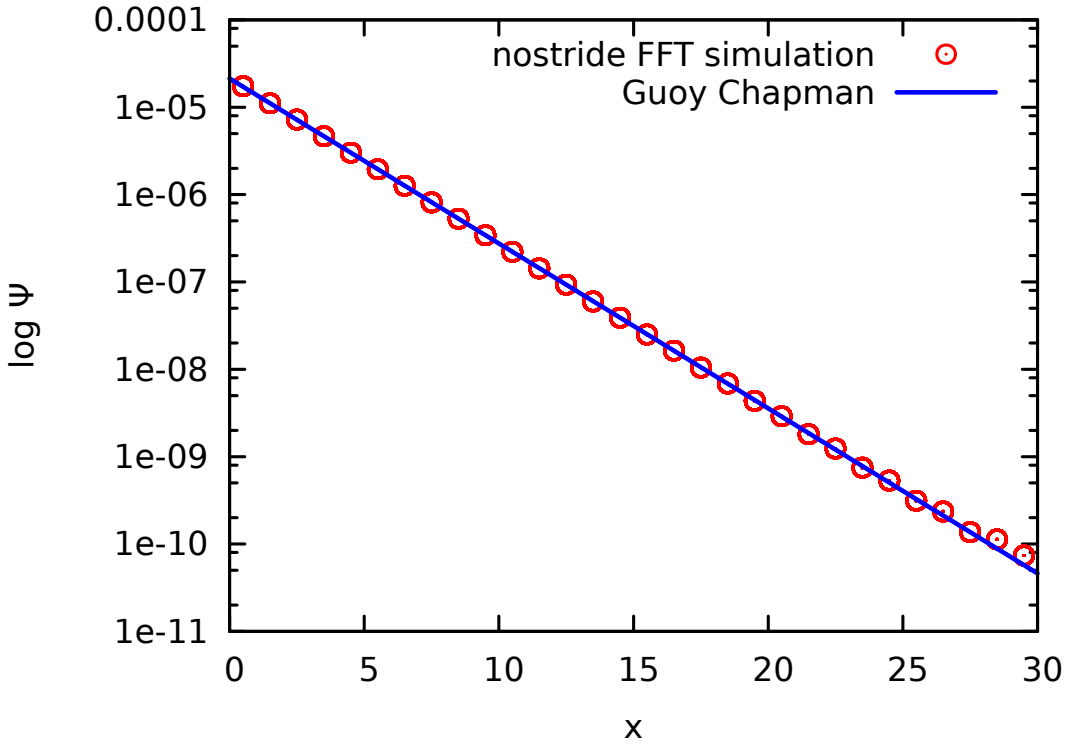
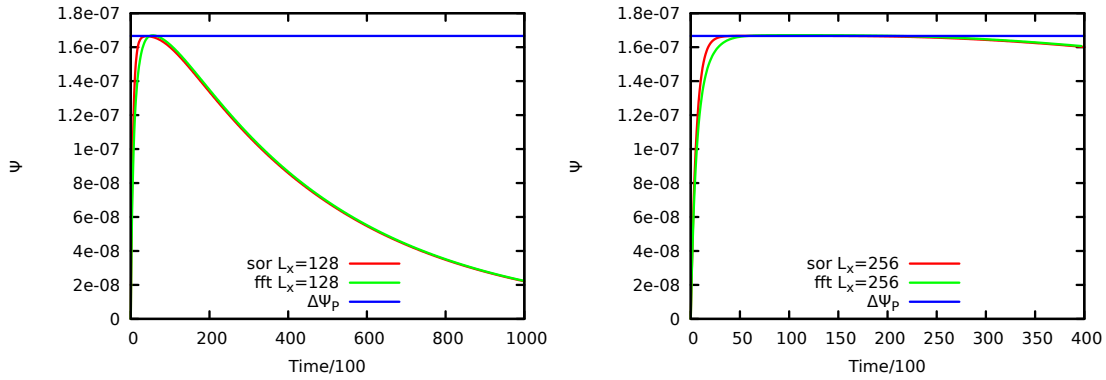
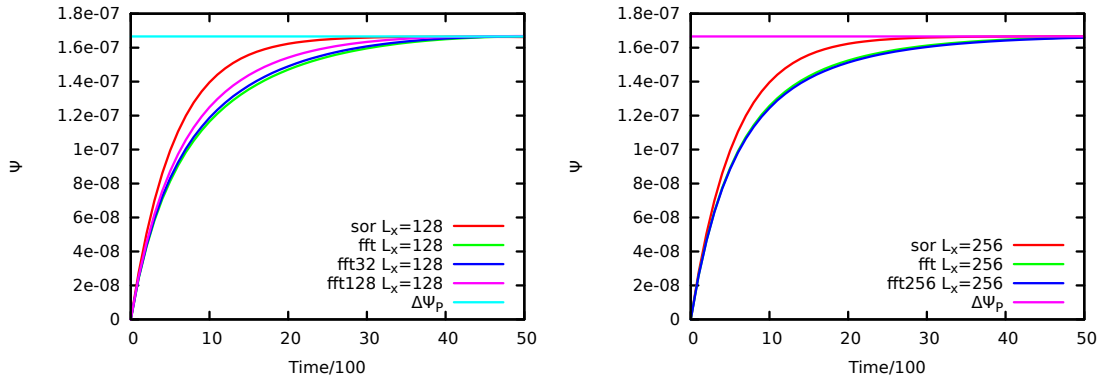


Figure 4.4: Results for the Gouy Chapman correctness test on HECToR when the `stride1` flag was not passed to the the P3DFFT library.

of this background term is obscured and it is expected that it would differ from method to method. When comparing with the theory, this background is subtracted as only the gradients of the potential are physically relevant. This also ensures that all values are above zero which is necessary to present this as a log plot.



(a) $L_x = 128$ *(b)* $L_x = 256$



(c) $L_x = 128$ *(d)* $L_x = 256$

Figure 4.5: Results for the liquid junction potential test on HECToR. Here we see the shape generated by the FFT solver is similar to that generated by the SOR solver. In the zoomed in plots, the sizes of the two other dimensions were varied and these sizes are indicated on the plot.

The bulk charge densities were found to be $\rho_{B,+} = \rho_{B,-} = 1.043 \cdot 10^{-2}$ for the SOR solver and $\rho_{B,+} = \rho_{B,-} = 1.044 \cdot 10^{-2}$ for the FFT solver. This also agrees with the original test of the SOR solver which found a value of $1.044 \cdot 10^{-2}$ for both bulk charge densities. The difference in the SOR solver results is due to the original simulation running on a $64 \times 2 \times 2$ grid, while this test was on a $64 \times 8 \times 8$ grid.

In the appendix, figure C.2 is also presented. This shows a plot of the SOR and FFT solver results for the Gouy Chapman problem against the analytical solution for the case where $\rho_{0,\pm} = 1 \cdot 10^{-3}$. Again we see close agreement with the theory suggesting that overall the FFT solver is producing acceptable results.

Presented in figure 4.4, is a plot of the potential for the Gouy Chapman problem when the P3DFFT library was built without using the `-enable-stride1` flag. This shows that the alternative k^2 multiplication implementation is also correct.

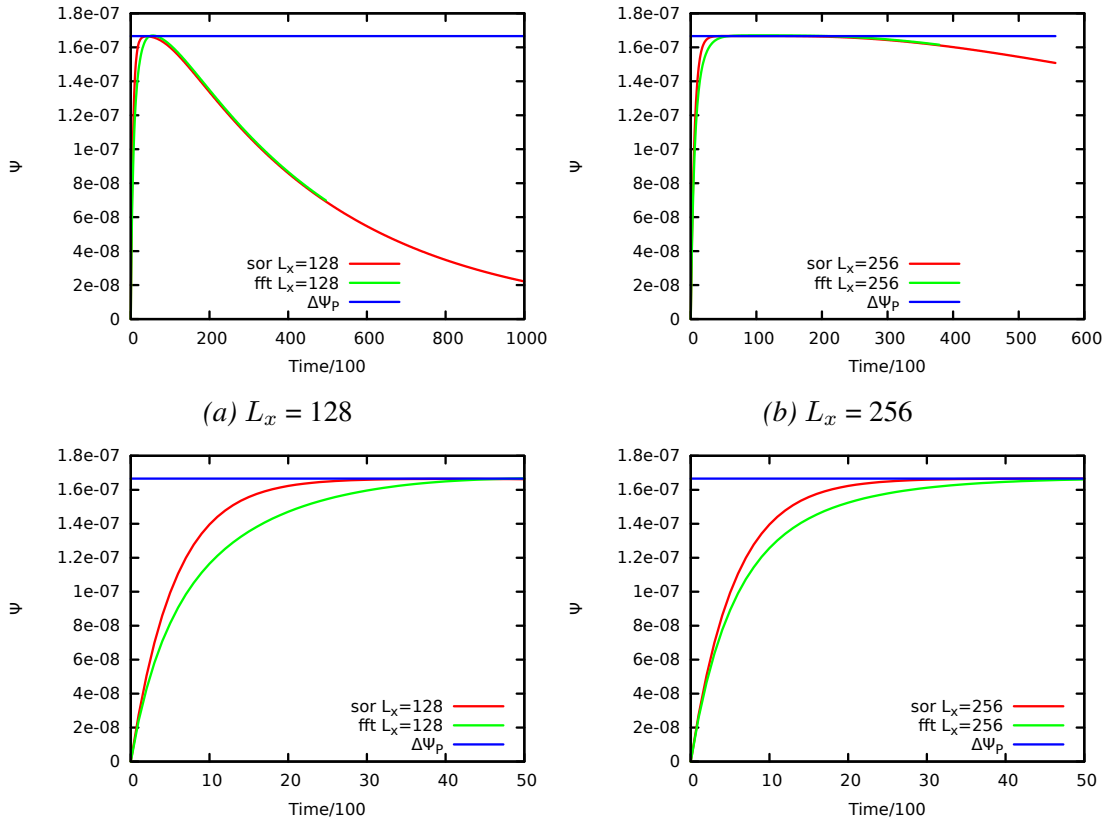
Using the `stride1` flag led to another slightly different value for $\Psi_c = -2.3289 \cdot 10^{-6}$. This shows how even a small change to the algorithm effects the background value,

while retaining the correct physical answers.

Overall, the FFT solver was found to provide excellent agreement with the theoretical answer to the physical Gouy Chapman problem. The project could thus continue to look at the liquid junction potential problem.

4.1.3 Liquid Junction problem

In order to check the correctness of the FFT solver in the liquid junction problem, plots were produced which show the evolution of the mean potential over time. These are shown in figures 4.5 and 4.6.



(a) $L_x = 128$ *(b)* $L_x = 256$

(c) $L_x = 128$ *(d)* $L_x = 256$

Figure 4.6: Results for the liquid junction potential test on BlueGene/Q. Here we see the shape generated by the FFT solver is similar to that generated by the SOR solver.

From these plots we see that the overall shape of the curves are quite similar and there is no difference between the results on HECToR and BlueGene/Q. There is however a slight deviation from expected when using the FFT solver. We see the asymptotic value reached is correct, but the number of time steps taken to reach it is slightly different.

It was during these tests that the problem of using long thin systems was first discovered. If a problem size of $128 \times 2 \times 2$ was used the answer was completely incorrect.

Increasing this to $128 \times 4 \times 4$ improved the result significantly. On HECTOR, using an even larger grid, such as $128 \times 32 \times 32$, was tested. This moved the answer slightly closer to the SOR solution. A cubic grid of 128^3 shows even closer agreement to the curve obtained by the SOR solver. These differences are only seen in the initial phase of the problem and the decay lengths are similar.

When testing the $L_x = 256$ system, a grid of $256 \times 4 \times 4$ was used for the SOR solver. This does not give the correct answer with the FFT solver. The smallest grid that could be used was one of size $256 \times 32 \times 32$. A larger 256^3 grid was also tested on HECToR to determine how much of an influence the grid sizes have on the accuracy. In this example there was little difference between the grid sizes. This is due to the fact that 32 is a large enough value to produce acceptable results.

Overall the results for FFT solver in the liquid junction potential problem show agreement with the SOR solver and thus the theory. It is posited that the difference in the boundary values cited in section 4.1.1 is the source of the difference between the SOR and FFT solvers. From this it can be deduced that the FFT solver will give reasonable physical approximations in further simulations.

4.2 Performance

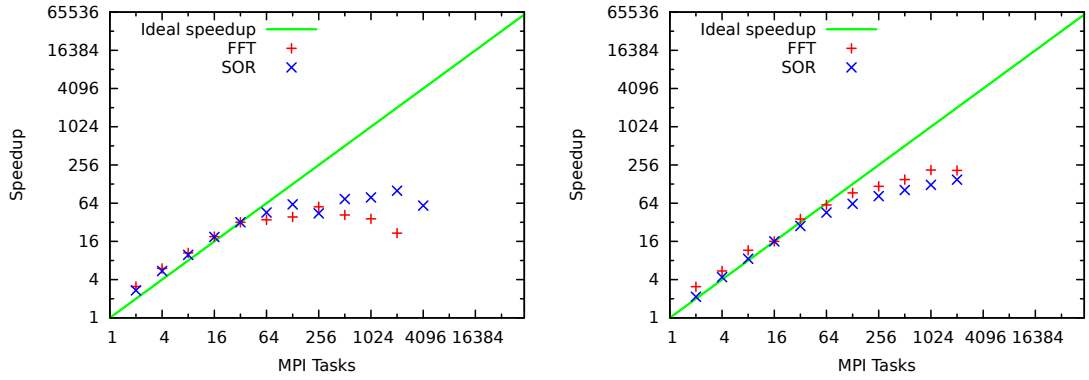
Now the results of the performance and timing testing will be presented. In all cases the FFT solver was found to produce results much faster than the SOR solver, though the scaling of the FFT solver on HECToR has not improved over the SOR method. Some differences in performance were found between HECToR and BlueGene/Q and these are mostly down to the difference in processor speeds and communications networks.

4.2.1 Gouy Chapman problem

An immediate performance gain was seen when using the FFT solver over the SOR solver. For the Gouy Chapman problem, with a $64 \times 8 \times 8$ (4096 total points) grid, on 8 cores the FFT solver averaged 0.00017 s for each update of Ψ. On the other hand, the SOR solver averaged 0.0145 s for the same problem. This is a speed up by a factor of over 100. The improvement in time comes from the reduction in the number of operations that must be performed. Using equations 2.3 and 2.13 we can look in more detail at the actual number of operations that must be performed. For this problem the SOR solver took 200 iterations for one solve. This equates to a total of $6 \times 4096 \times 200 = 4915200$ operations. The FFT solver takes $2 \times 4096 \times \frac{log(4096)}{log(2)} + 4096 = 102400$ operations. In theory this could show a speedup of up to 200 times. Swapping between decompositions and the all-to-all communications limit this to only 100 times speedup. The k^2 multiplication has also only been roughly accounted for and in reality the FFT algorithm requires more operations than this.

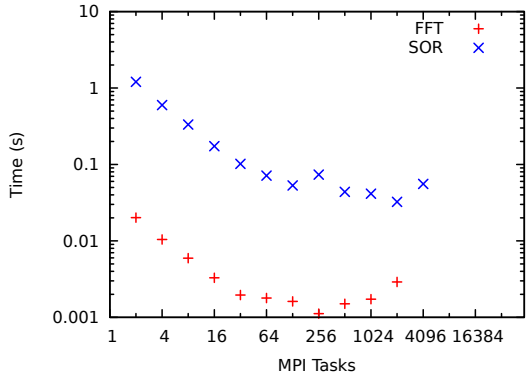
4.2.2 Liquid Junction Problem

In this section we see more performance increases when using the FFT solver instead of the SOR solver. The issue of scalability has not been solved with this new method on HECToR, but using the FFT solver is significantly faster than the SOR solver. However, BlueGene/Q shows excellent scaling for the FFT solver, surpassing that of the SOR solver in some cases and improving on the times for the FFT solver on HECToR on high cores counts.

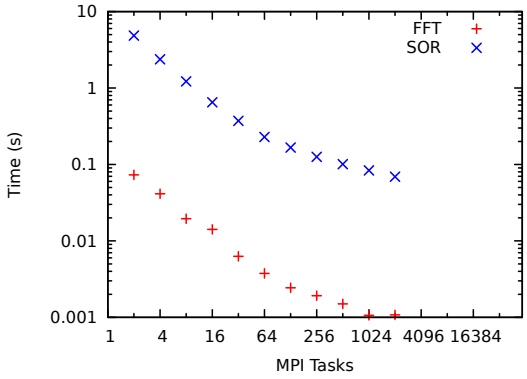


(a) HECToR - 64^3 speedup

(b) BlueGene/Q - 64^3 speedup



(c) HECToR - 64^3



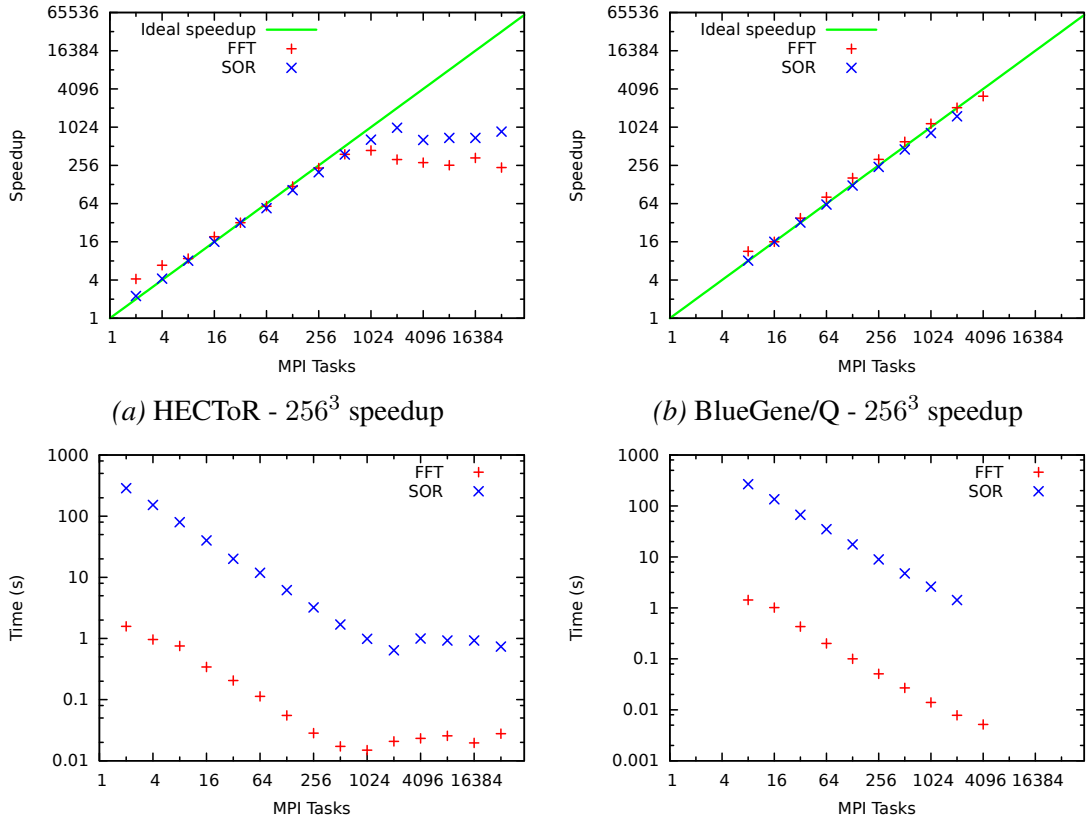
(d) BlueGene/Q - 64^3

Figure 4.7: Speedup and actual time per solve for the FFT solver and the SOR solver. This used a problem size of 64^3 for the liquid junction problem on HECToR and BlueGene/Q. The SOR solver took an average of 190 iterations to solve this system. Speedup is referenced to the number of cores in a node.

In figure 4.7 we can see the various measurements that were made for the 64^3 problem size on HECToR and BlueGene/Q. The speedup in these plots has been computed relative to the number of processors per node in the respective machines. The first thing that one notices is that the speedup of the SOR solver has not been improved upon and that HECToR has worse speedup than BlueGene/Q. However the actual time per solve has been improved upon significantly, by almost a factor of 100 on both machines.

It is interesting to compare HECToR and BlueGene/Q for this problem. We see that on

a low number of cores the performance of HECToR is much better. As the core count is increased, BlueGene/Q improves and even surpasses HECToR. The poor performance at low core counts is because BlueGene/Q has slower cores than HECToR (1.6GHz vs 2.3GHz). On the other hand, on paper, BlueGene/Q has better network capabilities. This is shown to be true by the performance of the FFT solver on larger numbers of cores as 3D FFTs are ultimately limited by communication.



(a) HECToR - 256^3 speedup

(b) BlueGene/Q - 256^3 speedup

(c) HECToR - 256^3

(d) BlueGene/Q - 256^3

Figure 4.8: Speedup and actual time per solve for the FFT solver and the SOR solver on the liquid junction problem. This used a grid size of 256^3 for the liquid junction problem on HECToR and BlueGene/Q. Speedup is referenced to the number of cores in a node.

In figure 4.8 we see that BlueGene/Q shows excellent scaling on the 256^3 problem, but slower performance on low numbers of cores. For this size of problem we see again that for large core counts BlueGene/Q can actually be faster than HECToR due to this scaling. This turnaround occurs at 1024 MPI tasks. BlueGene/Q takes 0.014 s per solve while HECToR takes 0.015 s. With higher numbers of tasks, BlueGene/Q continues to improve, while HECToR has reached its optimum performance.

The increase in problem size means that each processor has significantly more work to do and this affects BlueGene/Q more strongly for low core counts as it has the slower processors. Looking at the performance on HECToR we are interested to see how the

FFT solver compares relative to the SOR solver on larger problems. We see that, on 2 cores, increasing the problem size to 256^3, the FFT solver only slowed down by a factor of about 100. The SOR solver performed approximately 400 times slower. The computational complexity is dependent on the number of points in the system. Thus a lower complexity method shows more relative gains for larger problems and the FFT solver could be used for very large problems on machines with good communication networks.

When tested on the 128^3 problem size similar results were found as can be seen in figure C.3. BlueGene/Q's excellent scaling properties result in the time for solves improving over the times on HECToR above 512 MPI tasks. The unpredictability of communications can also be seen in the plot of the times on HECToR in this problem (figure C.3c). Times on 256 MPI tasks are slower than on 128 MPI tasks for both the SOR and FFT solvers in this case. It is likely that for this particular problem size and processor count the network becomes overloaded.

Times for the two solvers were also analysed for a problem size of 512^3. A plot of these times is available in the appendix, figure C.4. The FFT solver continues to show a speedup of a factor of 100 over the SOR solver. However, the performance on large numbers of cores is only about 10 times better on HECToR. In this problem the FFT solver must transmit more data, both in the decomposition switching stage and when computing the Fourier transforms. On large numbers of cores this is becoming a sizeable overhead. The performance is still better than that of the SOR solver because of the huge reduction in the number of actual operations that must be conducted. BlueGene/Q continues to show how good the interconnect is and the scaling seen is almost ideal. In this example it outperforms HECToR on core counts higher than and including 1024. If large problems are to be run with many cores, BlueGene/Q should be used over HECToR. Using small numbers of cores, HECToR is the recommended machine because of the improved performance and the shorter queue times.

Finally a problem size of 1024^3 was tested. This was done mostly as a proof of concept, that the FFT solver could solve this system in a reasonable time. Running this size is difficult, as it requires a large amount of memory and the output files created by Ludwig are also large. On HECToR with 2048 cores, each solve took 0.5 s. At this rate it would be possible to solve problems of this size, if the memory and storing of the resulting data were not issues.

One other thing noticed was the core counts that show the best performance on HECToR. These correspond to evenly decomposed pencil grids. For example we see peak performance on the 64^3 and 256^3 problems on 256 MPI tasks. This corresponds to a 16×16 pencil grid. The 512^3 problem shows peak performance on 1024 cores, or a 32×32 pencil grid This validates the proposal that the dimensions of the pencil grid should be kept as similar as possible.

In figure 4.9 the times taken for the SOR solver and the FFT solver with different size grids are presented. The SOR solver used a $256 \times 4 \times 4$ grid while the FFT solver used a $256 \times 32 \times 32$ grid. This plot was produced with the goal of determining whether the

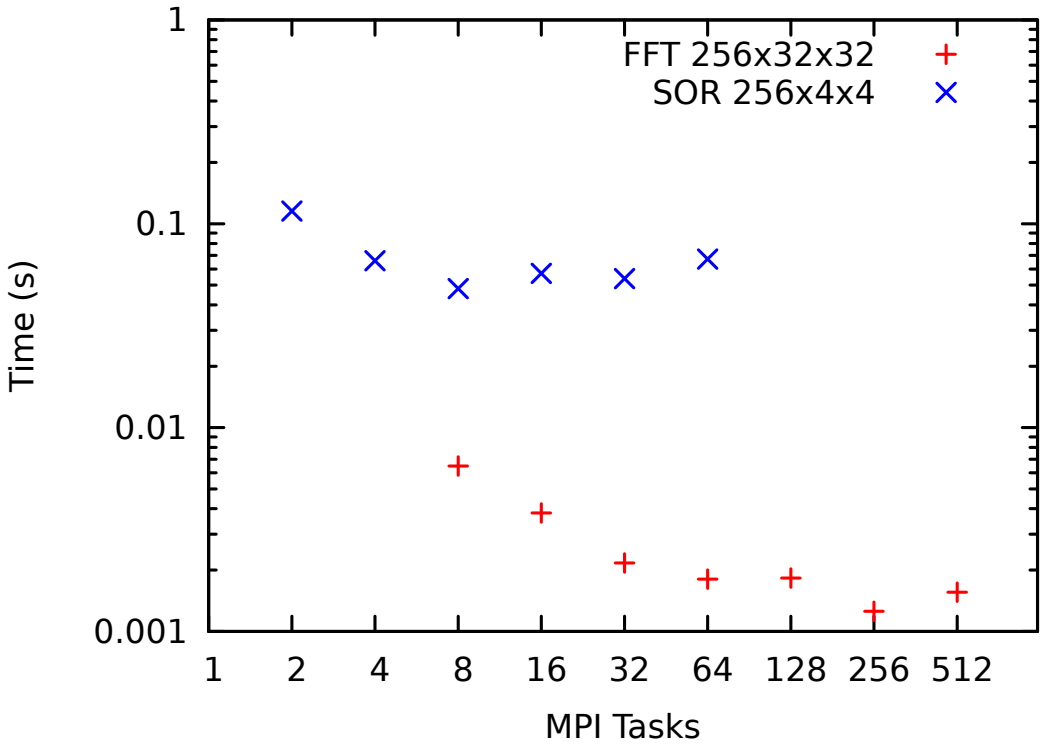


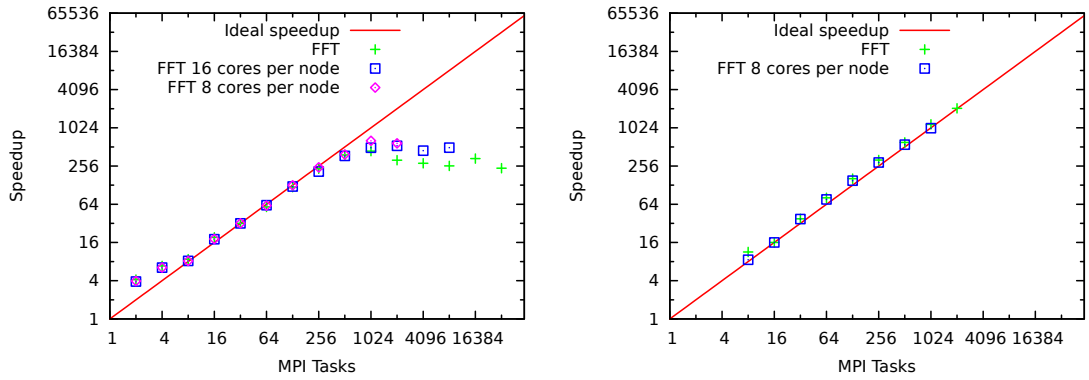
Figure 4.9: Plot of the times taken per solve of Poisson's equation for the SOR and FFT solver on differently sized grids on HECToR. The SOR solver used a $256 \times 4 \times 4$ grid, while a $256 \times 32 \times 32$ grid was used for the FFT solver.

FFT solver was actually a better method for solving a current physical problem. With the SOR problem size, there is a much lower limit on the core count that can be used. We are also concerned with the amount of budget on our machine that will be used up by solving this problem. On HECToR, where this particular experiment was conducted, there are 32 cores per node so the amount of time for this core count is important. As can be seen, the FFT solver clearly still solves the system much faster, even though it is larger. This shows that the constraint of not being able to use narrow systems is not a problem, and the FFT solver will still save the user time and money. It is expected that this will be the same on BlueGene/Q though it was not tested.

4.2.3 Under Subscribing Nodes

In order to look at other possibilities of improving the performance of the code, under subscribing nodes was briefly analysed. The speedup using 8 cores per node on BlueGene/Q, along with the speedup using 8 and 16 cores per node on HECToR are presented in figure 4.10.

We see here that there has been a very small performance benefit. This is due to the increased amount of cache and main memory available to the processors. The perfor-



(a) HECToR *(b)* BlueGene/Q

Figure 4.10: Plot of speedup on BlueGene/Q and HECToR for the 256^3 problem for under subscribed nodes. The original speedup with full node subscription is included for reference.

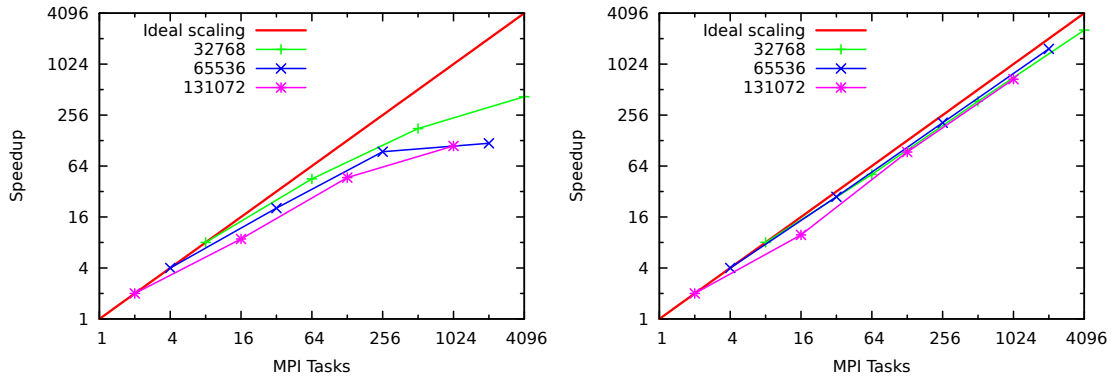
mance gain is such that it is not worth using this for real simulations as it would lead to significant increases in the amount of CPU time used. When nodes are under subscribed, the entire node is allocated and charged to the budget, which must be paid for. There are other issues with this as well. For example, on HECToR the cores on the node are filled from the start. This means that when using 16 cores in a node, they are all located on one processor[1]. If the user were to ensure that the cores were spread across these two processors it may be possible to show further performance increases. This was not deemed worthwhile for this project for budget reasons and due to it being a high risk endeavour, it is not likely that performance gains will be seen.

4.2.4 Weak Scaling

Now that it was clear that the FFT solver performed significantly better than the SOR solver, the weak scaling of the FFT solver could be analysed.

Figure 4.11 shows the weak scaling on both HECToR and BlueGene/Q with different size problem domains. When the grid size is increased the width and height of the domain on each processor must decrease to accommodate holding the entire depth of the problem. Were it the SOR solver we looked at, this would have a significant impact on the amount of communication taking place. When computing FFTs each processor must send almost the entire local data set to other processors. There will be more actual communications due to the increased number of processors and this will determine if communications are the limiting factor in this algorithm. The fact that the speedup tails off in figure 4.11a shows that this is the case. This is to be expected due to the heavy use of `MPI_Alltoall`. However, we again see that the network on BlueGene/Q is better than that of HECToR. Figure 4.11b shows almost perfect weak scaling on

[1]Since HECToR has two 16 core processors per node



(a) HECToR weak scaling

(b) BlueGene/Q weak scaling

Figure 4.11: Plots showing the weak scaling of the FFT solver on the BlueGene/Q and HECToR system for the liquid junction problem. The numbers show how many elements each processor had to work on. Th dimensions of the domains differ depending on the number of MPI tasks used as each must have entire rows of contiguous memory.

BlueGene/Q showing that the communication is not a large overhead on the calculation on this system.

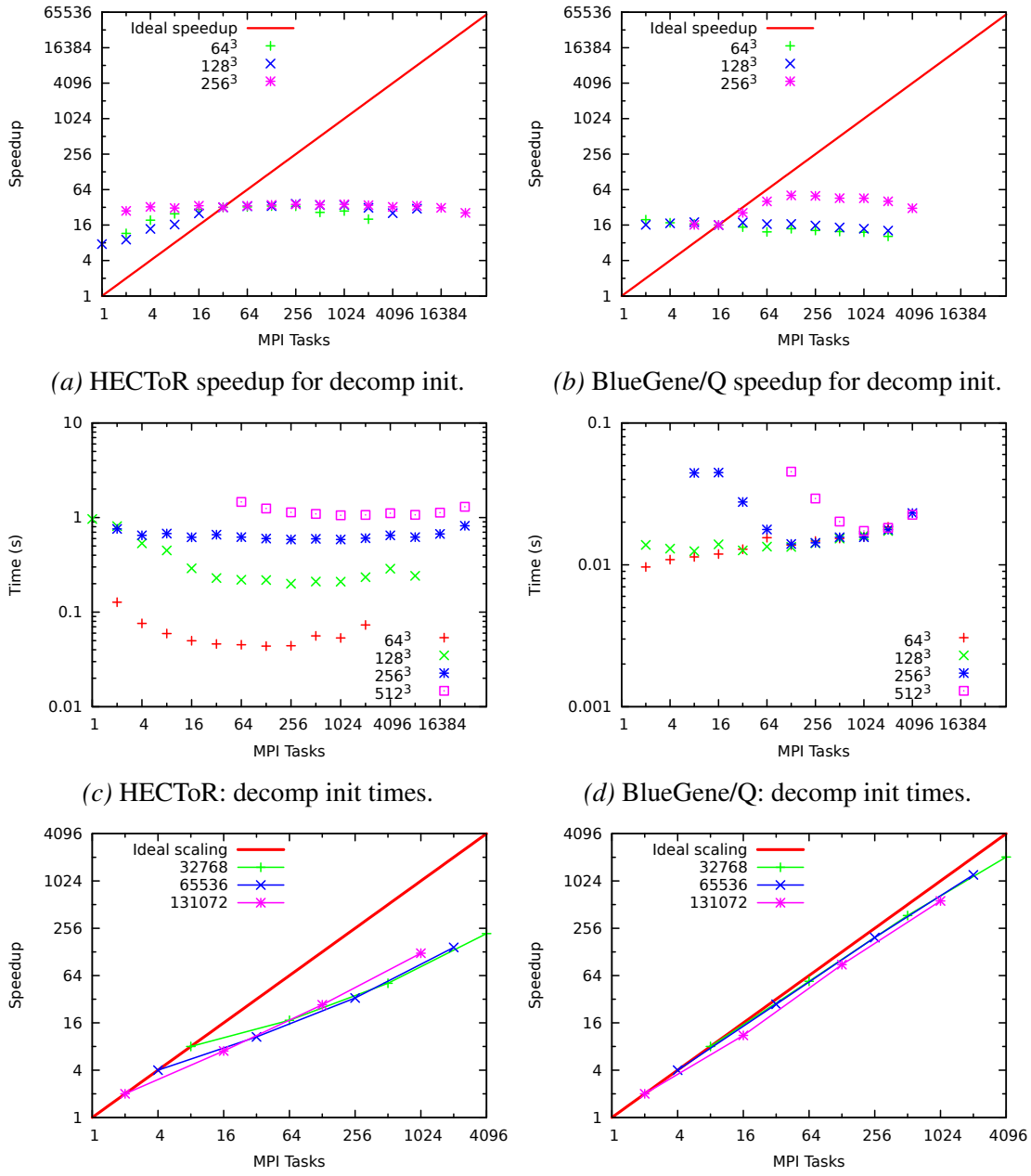
In light of this, in order to further improve the performance of the FFT solver, it would be worth considering reducing the communications. For example, it is possible that the decomposition switching routine could be further optimised.

4.2.5 Decomposition Initialisation & Switching

It was also of interest to determine if the decomposition switching routines were performing well. Plots of the various measured quantities when considering decomposition initialisation are presented in figure 4.12.

This initialisation routine shows unusual behaviour for a parallel program. On HECToR the times for this routine depend only on the grid size. Above 32 cores, for a given grid the time taken is roughly the same. In this routine only the first element of each subarray that needs to be sent is analysed. This saves on the actual number of elements to be processed. The number of messages being sent does not vary a largely with changes in processor count. Combined, these properties mean that changing the number of cores does not impact on the amount of work that each processor needs to do, so the time taken stays similar. Note that the messages being sent are the same size regardless of the grid size and the number of processors. This routine also makes calls to the P3DFFT library and it is likely that the timings on BlueGene/Q for large problem sizes and small core counts are due to these external routines. These involve the initial set up of the data and are dependent on the size of the problem and number of processors.

Comparing the times on HECToR and BlueGene/Q suggest that communication is the limiting factor in this routine. It would be expected that on low core counts HECToR's



(a) HECToR speedup for decomp init.

(b) BlueGene/Q speedup for decomp init.

(c) HECToR: decomp init times.

(d) BlueGene/Q: decomp init times.

(e) HECToR weak scaling.

(f) BlueGene/Q weak scaling.

Figure 4.12: Plots showing performance of the decomposition initialisation part of the code. In the first four plots numbers are the problem sizes. In the last two, the numbers are the number of lattice sites that each processor had in its local domain. Speedup is referenced to the size of a node.

faster processors would result in faster times than on BlueGene/Q. This is not the case and it would appear that BlueGene/Q's superior communications network is much more suited to this problem. It would be possible to conduct this routine without sending any messages. The result of this would be more computation, which could increase the time

taken on BlueGene/Q while reducing the time on HECToR. Since Ludwig is a code that will be deployed on many different architectures, the best overall performance should be selected. For this reason this routine could be modified to remove all communications.

This is not a particularly time critical piece of code, being called only once. We shall look at the 512^3 problem on 4096 processors on HECToR since this number of processors has the fastest time to solve Poisson's equation for this grid size. The time taken for one call to `psi_fft_poisson` is 0.06 s. The decomposition initialisation takes 1.11 s. After 20 updates of the potential, Ψ, more time has been spent on the update routine. This update routine will be called many thousands of times in an actual physical simulation, thus it is much more important that the update routine is fast.

Next, the actual switching time was measured. It was necessary to conduct these timings separately from the overall FFT routine timings to ensure the FFTs were timed correctly.

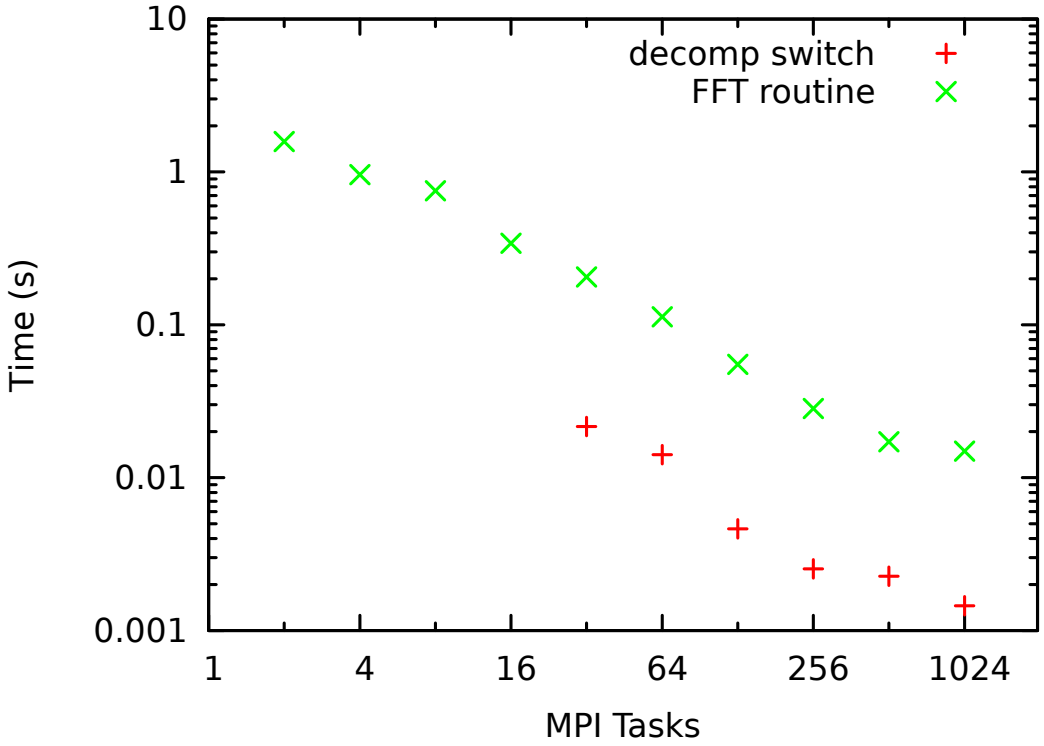


Figure 4.13: HECToR decomposition switching time. This plot compares the time for the total FFT solving routine with the amount of time switching between decompositions.

Figure 4.13 shows the time taken for the entire FFT routine to update Ψ and the actual switching of decompositions. Comparing these times it is clear that the decomposition switching is not the main time consuming part of the code. We know that the k^2 multiplication is O(N), while the FFT computation is O($NlogN$). Thus, in order to make further improvements in the overall speed of the computation, the actual FFTs would need to be modified.

Processors	Time (s) - `stride1`	Time (s) - No `stride1`
32	0.205	0.194
64	0.113	0.106
128	0.056	0.052
256	0.028	0.027
512	0.017	0.019
1024	0.015	0.015

Table 4.1: Table of the times taken per solve for the FFT solver on the 256^3 liquid junction problem with and without the `stride1` flag specified. These timings were conducted on HECToR.

4.3 Other Results

Further to the results found, some possible improvements to the code and implementations were proposed. There was a small amount of time left at the end of the project. It was decided to use this to test the performance of the `stride1` flag.

4.3.1 P3DFFT `stride1` flag

The effect of the `stride1` flag was also investigated for the Gouy Chapman and junction potential problems on HECToR. This flag is recommended by the author of the P3DFFT library and the code was thus built using this as there was not time provisioned for testing this at the start of the project.

For the Gouy Chapman problem it was found that not specifying the `stride1` flag had the effect of further improving the performance of the code. With the flag, the time per solve was $1.7 \cdot 10^{-4}$ s on 8 cores but without it, the time reduced to $1.4 \cdot 10^{-4}$ s. For the liquid junction problem there was also an improvement seen when this flag was not specified. As an example the 256^3 problem on 128 cores, the time reduced from 0.055 s to 0.052 s. A more comprehensive set of results can be seen in table 4.1. These improvements are small on a per solve basis, but, as obtaining physical results takes thousands of solves, it could be worthwhile availing of them. For example in the Gouy Chapman problem, where Poisson's equation is solved $40,000$ times to reach equilibrium, the overall time reduction in time spent solving Poisson's equation is 1 s, from 6.8 s to 5.8 s. In reality theses time savings are inconsequential considering the time taken to rebuild the library if the user would like to change the option. P3DFFT provides other options with the intention of improving performance. There was not time available to test any of these but it should be noted for future users.

Chapter 5

Discussion

In this chapter an analysis of the original risk assessment and work analysis will be undertaken along with a critical evaluation of the work done. Overall this project was seen as a success. The work plan and risk assessment proved useful in completing the project, though some adjustments were necessary.

5.1 Work plan

The importance of design was noted throughout the course of this project. The original work plan was created on the basis of an incomplete design, which led to it being hard to follow. This original plan can be seen in the appendix, section D.1. No time had been allowed for the creation of the decomposition switching routine as this had not been identified as necessary when preparing the project. This routine took time to write and to ensure correct functionality, as it was of vital importance. As a result the project time scale was set back immediately and time was not available for tuning the performance of the code on the two architectures used. The possibility of the FFT routine being hard to integrate with Ludwig was identified in the original risk assessment and the mitigation was applied successfully by shifting the work schedule and removing the performance tuning that had been planned for the end of the project.

Also of note is the fact that the boundary differences between the FFT and SOR solvers was not identified earlier in the project. Testing had been conducted but this was not rigorous enough, leading to delays later on. In this case these delays were not significant and provided more discussion of the method and algorithm used. It could have been the case that these errors led to significantly more incorrect results, with the impact of invalidating the results obtained. Much has been learned about design and the formulation of the work plan and this will be applied in future projects.

5.2 Risk Analysis

In terms of the original risk assessment, available in the appendix in section D.2, only one of the risks that were identified occurred as outlined above. One risk that was not predicted did occur towards the end of the project - the budget for CPU time on HECToR ran out. Another account with more budget had been set up earlier in the project and later timings could be conducted using this account. As a result no time was lost and more results could be found. However, this should have been identified as a risk with a large number of students using the same budget for an extended period of time.

Chapter 6

Conclusions

As a result of this project, Ludwig has a functional FFT solver for Poisson's equation in the electrokinetics module. This was the main goal of the project and has been achieved successfully.

In addition to this, timings were conducted that showed this FFT solver to be far superior to the original SOR solver in terms of time taken for each solve. However, The FFT solver was found to be inaccurate on small grids. This can lead to unpredictable behaviour in small or long, thin systems. For this reason the FFT solver in its current state could not completely replace the SOR solver. However, the FFT solver does have the advantage of, in theory, solving systems that are not electroneutral, which is a requirement of the SOR solver. This could lead to new problems being simulated.

Overall, the code necessary for the FFT solver is more complicated than that of the original SOR solver. Along with this, the program is also harder to use. There are more options in the input file that the user must check. The use of two libraries in order to conduct the FFTs makes the initial set-up of the program more difficult than previously necessary. Considering these overheads, the FFT solver is still seen as being superior to the SOR solver, though there are some improvements that could be made.

The library chosen, P3DFFT, was found to perform well but was not as user friendly as would be desired. The problems with installation on BlueGene/Q were the major drawback of this library. It cannot be guaranteed that it will work as packaged on any architecture. For the general user of Ludwig, it will be a large overhead in starting to use the code. Also, the fact that it is a Fortran library has an impact on the readability of the code in Ludwig. Arrays must be addressed using Fortran ordering, increasing the complexity of the code. A library written in C would be a significant improvement.

6.1 Further Work

With the current method, new physical systems can be modelled in a shorter time than before. In the future, a number of steps could be taken to improve on what has been done in this project. The decomposition initialisation could be done without communication to reduce the time it takes. It was also noted that the first Cartesian to pencil decomposition switch could be avoided by constructing the electric charge in the pencil decomposition. Writing a non-library Fast Fourier transform routine would also be of benefit to the users of Ludwig, allowing for the entire code to be available in one package and easily installable.

Alternatively, more advanced methods, such as Multigrid, which are O(N) could be implemented to further improve the speed and accuracy of solving Poisson's equation. While this would be difficult to implement it is likely that the performance gains would make it worthwhile.

Appendix A

Information on Ludwig

A.1 SVN Revision details

Ludwig is a project on CCPForge. This project was conducted on the branch fft-testcharge. This was branched from the testcharge branch at the start of the project. The last svn revision that is relevant to main body of this project is 2098. This is the revision that timings were conducted on. The code submitted is from revision 2141. This version is better formatted but there is little to no difference in the times.

A.2 File Changes

The following are the files that were modified in the branch:
Makefile: added flags necessary to compile executable with P3DFFT on both Blue-Gene/Q and HECToR.

ludwig.c:
included psi_s.h
changed psi_sor_poisson to ludwig->psi->psi_poisson_func
added call to decomp_finish if using psi_poisson_fft

psi_s.h:
added psi_poisson_func to psi_s

psi.h:
added new function psi_solver_set

psi.c:

added new function psi_solver_set

psi_rt.c:
added call to psi_solver_set
added call to info to print whether or not FFT solver will be used

timer.h:
added new timer id, TIMER_PSI_UPDATE

timer.c:
added a new timer name, psi_update.

input.ref:
added pencil_grid option and electrokinetics_fft option

The Makefile in tests was also edited to contain the necessary flags and new files.

A.3 File Additions

The following files were added to the src/ directory in the fft-testcharge branch:
psi_fft.h
psi_fft.c
decomp.c
decomp.h

The following files were added to the tests/ directory in the fft-testcharge branch:
test_psi_fft.c
test_decomp.c

Appendix B

Compiler flags

B.1 BlueGene/Q compiler flags

A list of the flags, and directories for finding those flags, necessary to compile C code invilving calls to the P3DFFT library on Bluegene/Q follows:

```
-L/bgsys/drivers/V1R1M2/ppc64/comm/xl/lib
-L/bgsys/drivers/V1R1M2/ppc64/comm/sys/lib
-L/bgsys/drivers/V1R1M2/ppc64/spi/lib
-L/opt/ibmcmp/xlsmp/bg/3.1/bglib64
-L/opt/ibmcmp/xlmass/bg/7.3/bglib64
-L/opt/ibmcmp/xlf/bg/14.1/bglib64
-R/opt/ibmcmp/lib64/bg/bglib64
-L/bgsys/drivers/toolchain/V1R1M2/gnu-linux/lib/gcc/
powerpc64-bgq-linux/4.4.6
-L/bgsys/drivers/toolchain/V1R1M2/gnu-linux/lib/gcc
-L/bgsys/drivers/toolchain/V1R1M2/gnu-linux/lib/gcc/
powerpc64-bgq-linux/4.4.6/../../../../powerpc64-bgq-linux/lib
-lmpich -lopa -lmpl -lpami -lSPI -lSPI_cnk
-lpthread -lrt -lstdc++ -lmpichf90 -lxlf90_r
-lxlopt -lxlomp_ser -lxl -lxlfmath -ldl
-lnss_files -lnss_dns -lresolv -lm
```

It is also worth noting, the ESSL library used is located at:
`/opt/ibmmath/lib64/libesslbg.a`

Appendix C

Results & Tables

C.1 Extra Plots

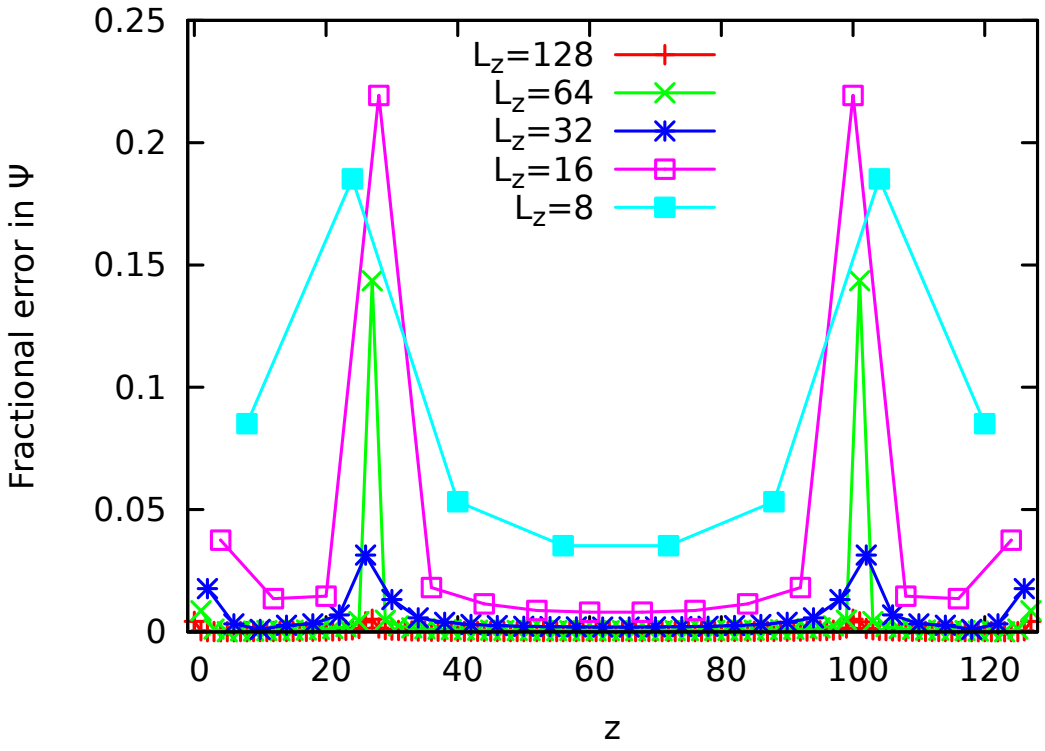


Figure C.1: Plot of the fractional difference between the SOR solver and the FFT solver for problem sizes from $L_z = 16$ to $L_z = 128$. The z positions have been scaled so all data sets will fit onto the same plot. This shows how the error becomes more prevalent with smaller sizes. The points of large error near $z = 20$ are where the potential is near zero, so computing the fractional error requires dividing by a very small number. A plot with these points removed can be seen in the results section in figure 4.2.

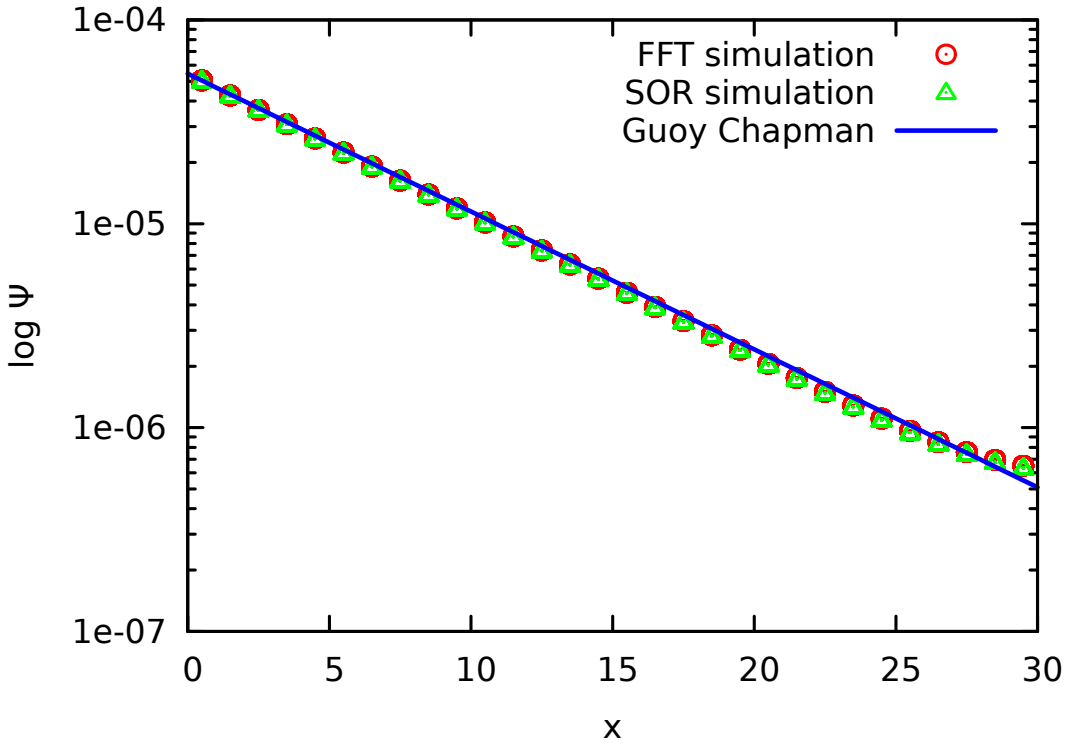


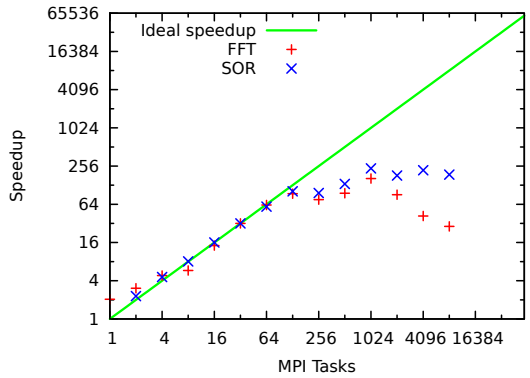
Figure C.2: Results for the Guoy Chapman problem on HECToR. The plot compares the simulation results when using the FFT solver and the SOR with the analytical solution for $\rho_{0,\pm} = 1 \cdot 10^{-3}$.

C.2 Experimental Parameters

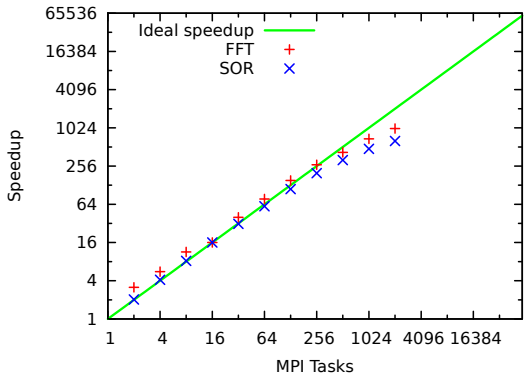
Here we present lists of the simulation parameters that were used in the experiments in order to aid repeatability. All values are given in simulation units.

C.2.1 Gouy Chapman problem

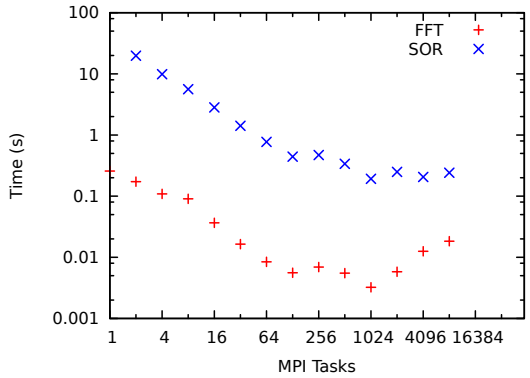
Unit charge $e = 1$
Temperature $k_B T = \beta^{-1} = 3.333 \cdot 10^{-5}$
Valency $z_\pm = \pm 1$
Dielectric permittivitty $\epsilon = 3.3 \cdot 10^3$
Diffusivities of the species $D_\pm = 1 \cdot 10^{-2}$
Initial charge densities $\rho_{0,\pm} = 1 \cdot 10^{-2}$
Bjerrum length $l_B = 0.723$
Initial surface charge $\sigma = 3.125 \cdot 10^{-2}$
The system was overall electroneutral.



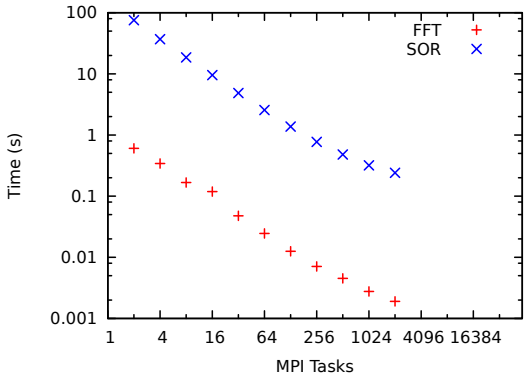
(a) HECToR - 128^3 speedup



(b) BlueGene/Q - 128^3 speedup

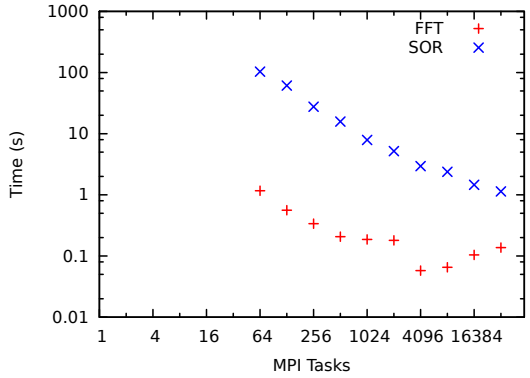


(c) HECToR - 128^3

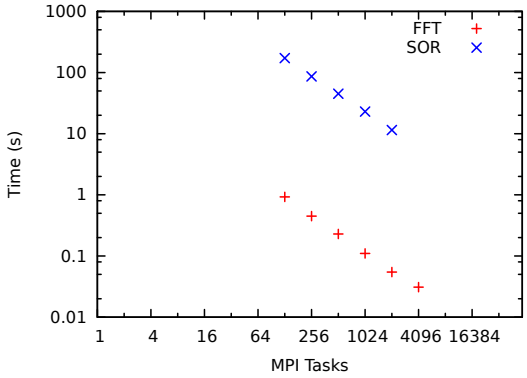


(d) BlueGene/Q - 128^3

Figure C.3: Plots of the speedup and actual solve times for a liquid junction problem size of 128^3 on HECToR and BlueGene/Q. Speedup is referenced to the number of cores on a node.



(a) HECToR 512^3



(b) BlueGene/Q 512^3

Figure C.4: Times for FFT and SOR solvers on the 512^3 liquid junction problem.

C.2.2 Liquid Junction Potential Problem

Unit charge $e = 1$
Temperature $k_B T = \beta^{-1} = 3.333 \cdot 10^{-5}$
Valency $z_{\pm} = \pm 1$
Dielectric permittivitty $\epsilon = 3.3 \cdot 10^3$
Diffusivities of the species $D_+ = 0.0125$, $D_- = 0.0075$

Appendix D

Workplan and Riskanalysis

D.1 Workplan

1. **Extracted FFT solver**
   The SOR solver part of LUDWIG will be extracted and an original implementation of the FFT solver built using this extracted version. This allows for easier development, with debugging and functionality tests being done on the actual solver instead of the entire LUDWIG code.

2. **FFT solver in LUDWIG**
   At this stage the FFT solver can be merged into the original code base using the knowledge gained from the extracted implementation.

3. **Tests of Functionality**
   While tests will be running throughout to ensure the implementation is correct, LUDWIG includes some specific tests. This stage will involve ensuring these tests pass and the answer obtained by the new implementation is correct.

4. **FFT & SOR Benchmarks**
   In order to judge the the level of success of the project it will be necessary to test the performance of the new FFT implementation against the old SOR one. Of importance to test will be the relative performance in terms of compute time and scalability of the solution.

5. **Analyse Results**
   Now it will be possible to look at the data from the tests of the SOR solver against the FFT solver and conclude whether there has been a performance increase, and if there is the value of this.

6. **Performance Tuning HECTOR**
   Since HECToR is one of the target architectures for LUDWIG, it will be worthwhile taking time to ensure performance is maximised for the architecture. There are a number of factors that could impact on the performance of the code, such as

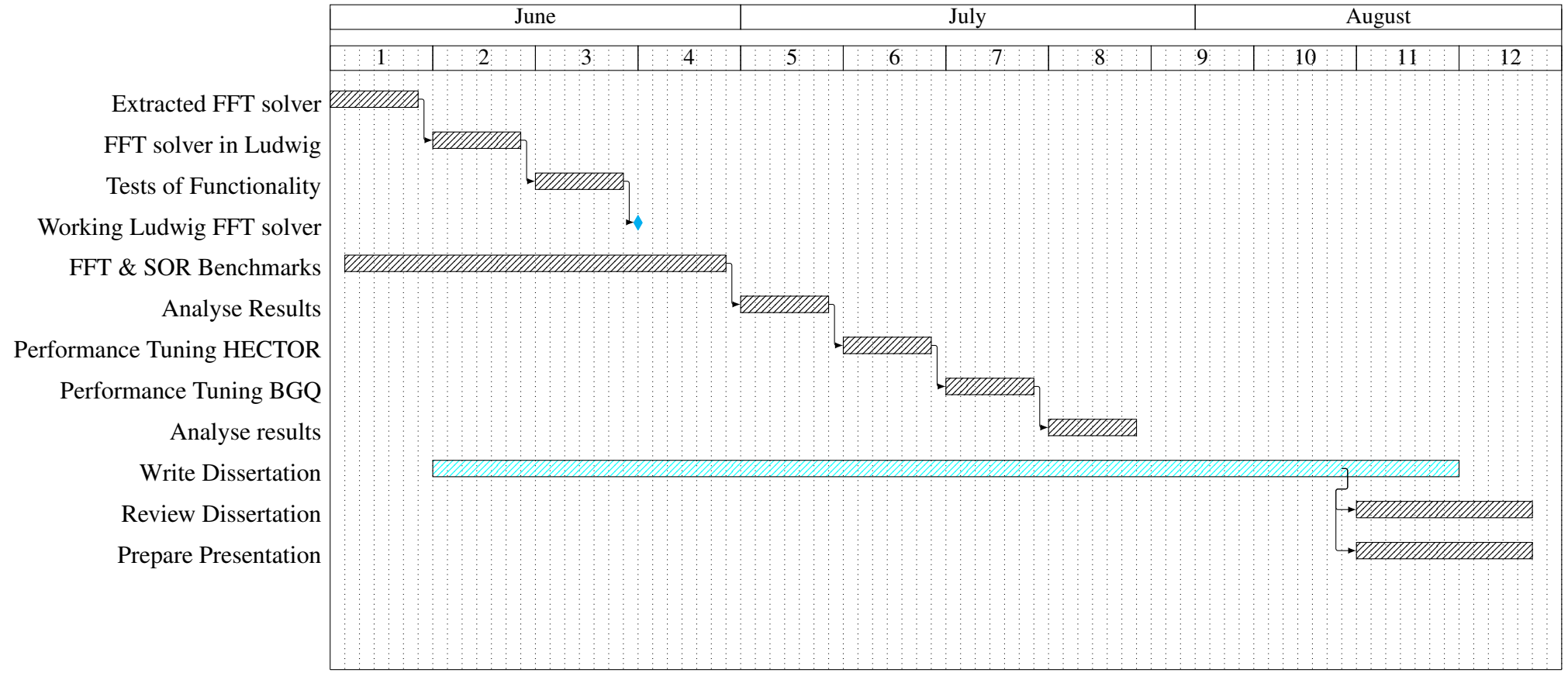
how P3DFFT is built, and these will need to be investigated.

7. **Performance Tuning BG/Q**
   The BlueGene/Q machine will also be available for this project and it will be interesting to compare performance on this against HECToR. This is obviously time permitting and will depend on whether the work goes to plan up to this stage.

8. **Analyse results**
   Once again, after the performance has been tuned

9. **Write Dissertation**
   The intention here is that this will be an ongoing process throughout the period. This should serve to reduce pressure at the end of the work period and allow for more time to be spent reviewing the writing.

10. **Prepare Presentation**
    Most of the material for this will already have been generated during the writing of the dissertation. However, it will be necessary to ensure there is time available to practice the presentation so that the requirements, such as length, are met satisfactorily.


June
July
August
1
2
3
4
5
6
7
8
9
10
11
12
Extracted FFT solver
FFT solver in Ludwig
Tests of Functionality
Working Ludwig FFT solver
FFT & SOR Benchmarks
Analyse Results
Performance Tuning HECTOR
Performance Tuning BGQ
Analyse results
Write Dissertation
Review Dissertation
Prepare Presentation

D.2 Risk Analysis

Risk	FFT method not faster than SOR
Likelihood	Low
Mitigation	Have other methods that can be investigated. Conduct timings as soon as possible to allow maximum time for this
Impact	High (change to project outcomes)
Risk	Difficulty in getting a working solver of Poisson's Equation with FFTs
Likelihood	Medium
Mitigation	It may be difficult to have an FFT solver work correctly in conjunction with LUDWIG. Performance tuning can be neglected if this takes more time than expected.
Impact	Medium (2-3 weeks)
Risk	Selected for World Orienteering Champioships (week 6 of schedule)
Likelihood	Medium
Mitigation	Will budget extra time, and plan to have work that does not need internet access to do that week.
Impact	Low (1 week)
Risk	Technical difficulties such as P3DFFT installation on BG/Q problems
Likelihood	Low
Mitigation	Try to ensure all necessary libraries are installed on the systems to be used before the work on the project actually begins. Already installed on HECToR.
Impact	Low (1 week)
Risk	Lack of access to machines (HECToR and BG/Q)
Likelihood	Low
Mitigation	Ensure all work is in revision control and can use the Morar system for smaller scale tests. How much of a problem this is, is very dependent on the time during the project which it occurs.
Impact	Low (1 week)

Bibliography

[1] Arfken, G. *Mathematical Methods for Physicists*, **3**, Academic Press, pp. 485-486, 905, and 912, (1985)

[2] Gerhard Besold, *Derivation of the Poisson-Boltzmann equation*, (2006)

[3] Bluegene/Q Website *http://www.epcc.ed.ac.uk/facilities/dirac* (5th Aug 2013)

[4] IBM ESSL documentation (Online), *http://publib.boulder.ibm.com/infocenter/clresctr/vxrx/index.jsp?topic=i%2Fcom.ibm.cluster.pessl.v4r2.pssl100.doc%2Fam6gr_bbibli.htm* (6th Aug 2013)

[5] Evangelos Brachos, *Parallel FFT Libraries*, University Of Edinburgh, (2011).

[6] M.E. Cates, K. Stratford, R. Adhikari, P. Stansell, J.-C. Desplat, I. Pagonabarraga, and A.J. Wagner *Simulating colloid hydrodynamics with lattice Boltzmann methods*, J. Phys.: Condens. Matter **16**, S3903-S3915 (2004)

[7] Complex Fluid Lab Website, University of Huelva, *http://www.complexfluidlab.com/complex_fluid_eng.htm* (26 Jul 2013)

[8] J. Cooley and J. Tukey *An Algorithm for the Machine Calculation of Complex Fourier Series*, Mathematics of Computation, **19** 90, pp. 297-301, (1965)

[9] P. Debye and E. Hückel *The theory of electrolytes. I. Lowering of freezing point and related phenomena* Physikalische Zeitschrift **24** 185-206 (1923).

[10] M. Deserno and C. Holm, *How to Mesh up Ewald Sums*, J. Chem. Phys., **109**: 18, (1998)

[11] J.C. Desplat, I. Pagonabarraga, P. Bladon, *Ludwig: A parallel Lattice-Boltzmann code for complex fluids*, **134**, 3, 273-290, (2001)

[12] Frigo, Matteo and Johnson, Steven G., *The Design and Implementation of FFTW3*, Proceedings of the IEEE **3**, 2, pp 216–231 (2005)

[13] FFTW documentation (Online), *http://www.fftw.org/fftw3_doc/The-1d-Discrete-Fourier-Transform-_0028DFT_0029.html#The-1d-Discrete-Fourier-Transform-_0028DFT_0029*, (24 Jul 2013)

[14] Alan Gray, Alaistair Hart, Oliver Henrich and Kevin Stratford, *Scaling Soft Matter Physics to Thousands of GPUs in Parallel*, EASC (2013)

[15] L. Greengard and V. Rokhlin, *A Fast Algorithm for Particle Simulations*, Journal of Computational Physics, **73**, pp. 325-348 (1987)

[16] David J. Griffiths *Introduction to Electrodynamics* Reed College, 4th ed. (2013)

[17] HECToR Website *http://www.hector.ac.uk/* (5th Aug 2013)

[18] David Henty *Advanced Parallel Programming, Lecture 12, Parallel FFTs* University of Edinburgh, (2013)

[19] R.W. Hockney and J.W. Eastwood *Computer Simulation Using Particles*, ed.3, Taylor & Francis Group, pp. 18-24 (1988)

[20] Heike Jagode *Fourier Transforms for the BlueGene/L Communication Network* University of Edinburgh (2006)

[21] M. Kazhdan, M. Bolitho and H. Hoppe *Poisson Surface Reconstruction* Eurographics Symposium on Geometry Processing (2006)

[22] V.M. Kendon, J.-C. Desplat, P. Bladon, and M.E. Cates *3-D Spinodal Decomposition in the Inertial Regime* Physical Review Letters **83** pp. 576-579 (1999).

[23] N. Li and S. Laizet, *2DECOMP&FFT - A highly scalable 2D decomposition library and FFT interface*, Cray User Group 2010 conference, Edinburgh, (2010)

[24] Ludwig Documentation (2013)

[25] J. Lyklema *Fundamentals of Interface and Colloid Science* Academic Press (1995).

[26] S. Mafé, J.A. Manzanares and J. Pellicer, *The Charge Separation Process in Non-Homogeneous Electrolyte Solutions* J. Electroanal. Chem. **241**, 5 7-77 (1988).

[27] MareNostrum supercomputer website *http://www.bsc.es/marenostrum-support-services*

[28] I. Pagonabarraga, B. Rotenberg and D. Frenkel *Recent advances in the modelling and simulation of electrokinetic effects: bridging the gap between atomistic and macroscopic descriptions* Physical Chemistry Chemical Physics **12** pp.9566-9580 (2010).

[29] Dmitry Pekurovsky *P3DFFT: A Framework for parallel computations of Fourier Transformations in Three Dimensions*, SIAM J. Sci. Comput. **34**, 4, pp.C192-C209, (2012)

[30] William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery *Numerical Recipes in C* §12 Cambridge, 2, 1999

[31] William H. Press, Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery *Numerical Recipes in C* §19 Cambridge, 2, 1999

[32] B. Rotenberg, I. Pagonabarraga and D. Frenkel *Coarse-grained simulations of charge, current and flow in heterogeneous media*, Faraday Discussions: Multiscale Modelling of Soft Matter, **144**, pp. 223-243 (2009)

[33] Titan website http://www.olcf.ornl.gov/titan/ (5th Aug 2013)

[34] Ulrich Trottenberg, C. Cornelis W. Oosterlee and Anton Schulle *Multigrid* Academic Press, ed. 1 (2001)