Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Mathematics

Investigation into the Numerical Computation of the Laplace Heat Equations for 2D Conduction

Fionntán Ó Suibhne
20318778

September 2021

A thesis submitted in partial fulfilment
of the requirements for the degree of
MSc (Masters in High Performance Computing)

Declaration Concerning Plagiarism

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at http://www.tcd.ie/calendar.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at http://tcd-ie.libguides.com/plagiarism/ready-steady-write.

Student Number: *20318778* Signed: _____ Date: _____

Acknowledgements

I would like to thank my supervisor, ***Prof. Kyle Parfrey*** who provided valuable comments and feedback throughout the production of this body of work and hence added to the quality of this thesis. For this I thank you.

All calculations were performed on the Kelvin cluster maintained by the Trinity Centre for High Performance Computing. This cluster was funded through grants from the Higher Education Authority, through its PRTLI program.

I should acknowledge ***Darach Golden*** my HPC Software lecturer who provided a basis in much of the MPI concepts utilised and a range of sample code to demonstrate them, some of which informed the 'Layout and Software Design' section.

Finally, I would like to thank my parents, ***Domhnall Sweeney*** and ***Anne Marie Cleary*** for their untiring support through this period time.

Table of Contents

List of Tables of Figures

Nomenclature

M	$Rows$
N	$Columns$
T	$Temperature$ [℃]
T_m	$Max\ Temperature$ [℃]
a, b	$Cartesian\ Length$ $[m]$
x, y, z	$Cartesian\ spatial\ axes\ dimesions$
t	$time\ dimension$
k	$material\ thermal\ conductivity$ $[W/m℃]$
g	$Spatially\ varying\ internal\ heat\ generation$ $[W/m^2]$
α	$Material\ thermal\ diffusivity$ $[m^2/s]$
i, j	$spatial\ grid\ index$
λ	$Unknown\ constant\ Lambda$
v, w	$functions$
v_x	$first\ derivative\ of\ function\ with\ respect\ to\ x$
w_y	$first\ derivative\ of\ function\ with\ respect\ to\ y$
v_{xx}	$second\ derivative\ of\ function\ with\ respect\ to\ x$
w_{yy}	$second\ derivative\ of\ function\ with\ respect\ to\ y$
x, y	$Spatial\ distance\ from\ orogin$ $[m]$
i	$Imaginary\ number$
c_1, c_2, c_3, c_4	$Unknown\ constants$
A, A_1, A_2, A_3	$Unknown\ constants$
B, B_1, B_2, B_3	$Unknown\ constants$
C, C_1, C_2, C_3	$Unknown\ constants$
D, D_1, D_2, D_3	$Unknown\ constants$
$f_{i,j}$	$Finite\ difference\ representation\ of\ spatial\ function\ values$
$h_a,\ h_b$	Heat transfer coefficients [W/m^2℃]
$T_{\infty,a}, T_{\infty,b}$	Ambient Temperature {℃]
β_o, β_N	Biot constant
γ_o, γ_N	Gamma constant
G_o, G_N	Heat generation constant
Δx	Spatial step in Cartesian x axis
Δy	Spatial step in Cartesian y axis
Δt	Time step
r_{sx}, r_{sy}	Spatial dimension Stability Ratio
r_s	Total Stability Ratio Sum
$r, \emptyset, z$	$Polar\ spatial\ axes\ dimesions$
$\emptyset$	$Tangential\ distance$ $[radians]$
r	$Internal\ radius$ $[m]$
Δr	$radial\ change$ $[m]$
r_i, a	$Internal\ Annular\ radius$ $[m]$
r_o, b	$External\ Annular\ radius$ $[m]$
T_i, T_0	$Internal\ \ annulus\ boundary\ temperaure$ [℃]
T_o, T_N	$External\ \ annulus\ boundary\ temperaure$ [℃]
δ	$Radial\ step\ size$ $[m]$
C_1, C_2	$Unknown\ constants$
ΔT	$Temperature\ change$ [℃]
s	$serial\ fraction$
p	$Parallel\ fraction$
N, n	$Number\ of\ Processors$
t_1	$Serial\ execution\ time$ $[s]$
t_n	$Parallel\ execution\ time$ $[s]$

1. Introduction

Efficient energy storage systems play a vital role in the 21st century and will continue to do so for the foreseeable future be this in alleviating the strain of peak demand on electrical grids, compensating for renewable energy's intermittent production, supplying the growing demand for electric vehicles or smartphone technologies. Lithium batteries have held a dominant position commercially for their high energy density over the past 40 years [1]with improvements slowing as theoretical maximum capacities are approached. However an exciting rethink of the traditional manufacturing process, announced in a recent Tesla 'Battery Day' presentation, should in theory, allow for a significant leap in the efficiency of lithium cells [2].

Involving the use of a "tab-less" design, electrons within cells will have less distance to travel which should translate to a reduction of the internal resistance, typically responsible for energy losses through heat generation. It is for this reason that the ability to simulate the thermal performance of batteries is of great interest. The thesis will aim to use parallel computing techniques to implement the preliminary numerical simulation steps required for modelling the thermodynamic behaviour of battery like structures.

Due to the high complexity of the thermodynamic processes involved in battery technology and time constraints, significant simplifications will be made in order to narrow the projects scope down to practical and actionable targets. One such simplification will be the assumption that the battery behaves as a solid where only conduction applies with exception to boundary conditions.

2. Objectives

There were four primary objectives; the **first focused on the shape**, initially a 2D cross section, which uses a polar co-ordinate system to mesh **annulus style geometry**. This leaves open the potential for an additional dimension to form a hollow cylinder modelling a very typical cell shape such as the 18650 battery. The annulus shape holds the advantage over a disk in the avoidance of a singularity occurring at the centre (0,0). Although this can be overcome with application of an individualised expression applied at this node, it's a layer of additional complexity [3].

It may be necessary to apply such an expression in the case of future battery technology's which see the replacement of this hollow with a thermally conductive material in order to cool batteries axially through the core. The annulus geometry has no need for tangential boundary conditions due to its periodic nature however a **second objective** was to feature **convective boundaries** on the inner and outer radial edges.

The **third objective** was a **multi-dimensional time dependant** heat conduction simulation. The time dependency can be obtained through an explicit method with second order accuracy however comes at a cost as it imposes additional stability criteria in the relationship between the time step and the spatial steps. These objectives are all obtainable in theory however only covered in isolation from each other in resources [3] and it is assumed that a final model involving all three objectives is achievable. The code was to be written in C and parallelized using OpenMPI with no other significant libraries expected for meshing or decomposition.

The **fourth objective** was the implementation of multidimensional steady state system with **internal heat generation** which can be achieved simply with the inclusion of energy generation term to the heat equation.

The Finite Difference Method (FDM) was used to discretize the necessary partial differential equations as opposed to alternatives such as Finite Element Method (FEM) or Finite Volume Method (FVM), for its ease of implementation when applied to simple geometries and efficiency as a more direct approach. Jacobi iteration was used for its simple point iteration [3] with a specified tolerance as convergence criteria. This leaves the option of successive over relaxation as an additional possible project objective for inclusion.

3. Background

3.1 Finite difference Methods

Finite difference methods are an approach to solving partial differential equations (PDEs) such as those found in the Laplacian heat conduction formula through discretization. They offer a simple coherent process to solving PDEs of simple geometries such as those under consideration in this project and for that reason will be considered for use over similar alternatives like finite element modelling (FEM).

The numerical method is dependent on the partial differential equation under consideration; this in turn depends on the problem being solved. Whether internal heat generation is present, the number of dependant spatial axes and time dependency will impact the Laplace PDE required for solution. This report will focus on two PDE types; elliptic and parabolic. An example of an elliptic PDE is shown in (3.1-1) with a two dimensional steady state heat conduction problem derived from the general solution for the Laplacian. Elliptic problems require that all boundary conditions of a domain are known to solve.

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{1}{k} g(x,y) = 0 \qquad (3.1\text{-}1)$$

Similarly a parabolic example is shown (3.1-2) of one dimensional time dependant heat conduction derived from the general solution for the Laplacian. In this case a time dependency exists, however because the solutions of time are influenced by the past only and not future events, an initial time state is all that is necessary to allow for the solution of an unknown future state in time as opposed to a start and end time state. These classifications dictate the finite difference stencils to be used.

$$\frac{\partial^2 T}{\partial x^2} = \frac{1}{\propto} \frac{\partial T}{\partial t} \qquad (3.1\text{-}2)$$

A continuous function can be discretised into small enough step sizes such that the approximate differentials can be expressed as finite difference equations (3.1-3), (3.1-4) & (3.1-5) using the notation shown in Figure 3.1.1. These are but a few of the different forms of finite difference approximations but are related to the classifications mentioned previously for use in this project.

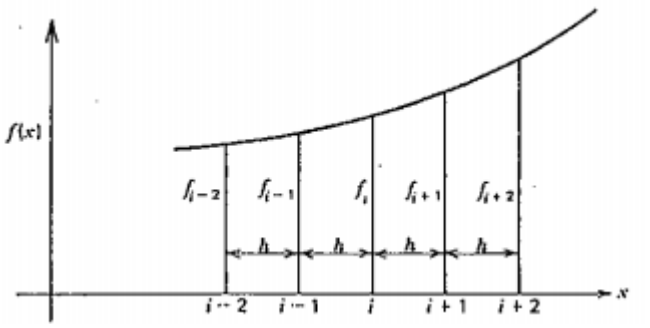

Figure 3.1.1: Finite difference representation of f(x) [3]

$$f_i' = \frac{f_{i+1} - f_i}{h} + O(h) \rightarrow \ forward\ difference \quad (3.1\text{-}3)$$

$$f_i' = \frac{f_{i+1} - f_{i-1}}{2h} + O(h^2) \rightarrow \ central\ difference \quad (3.1\text{-}4)$$

$$f_i'' = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2} + O(h^2) \ \rightarrow \ central\ difference \quad (3.1\text{-}5)$$

Equation (3.1-3) is the first order forward difference formula necessary for applying the parabolic classification where transient conditions exist in equation (3.1-2). This finite difference approximation is dependent on two grid points; referred to as a Two-point formula or stencil and as an approximation has an error on the order of h. Equation (3.1-5) is a second order three-point stencil of the central difference approximation required for the Elliptic situation expressed previously (3.1-1) and has an error on the order of h^2. The total error for finite difference calculations is expected to be minimized when the grid mesh size is reduced. [3]

3.2 Numerical Method for solving Algebraic Equations

The finite difference approximation allows the partial differential heat conduction formula to be expressed as a discretised domain of points with a set of boundary conditions. This still requires application of a method for solving the algebraic equations generated. This can be done using direct or iterative methods, each with their own advantages. While a direct method such as Gauss Elimination would be better suited to the solution of the problems posed due to the presence of banded matrix coefficients it was deemed more realistic to implement a simpler iterative method initially with the intension to investigate more complex and efficient methods later.

Iterative methods are used to solve the algebraic equations many times over, considering results from previous iterations to progressively converge on the solution. The greater the number of iterations the more accurately the solution can be approximated (assuming stability). The Jacobi method is not the most efficient method for convergence and requires a diagonally dominant coefficient matrix which makes it restrictive but it is "instructive and serves as a prototype for more advanced algorithms" [4], it was selected for its simple point iteration with a specified tolerance as a convergence criteria. In retrospect the Gauss-Seidel iteration should have been applied as is equally instructive yet converges to solutions much faster. [3]

3.3 Annular Geometry

Simulation of an annular geometry was applied and is explained in greater depth throughout this report however as an introduction, one can start with the general solution for the Laplacian shown in (3.3-1). This is applied to a cylindrical shaped object where r represents the radial axis, Theta ($\emptyset$) the tangential axis and z the spatial axial axis. The tangential axis is periodic with a period of 2π. Terms which are not relevant to the simulation under consideration can be removed allowing this formula to derive any variation of the dimension types. It is important to note however the dependency of the tangential dimension on the radial axis which remains even in situations where heat conduction is not simulated in the radial dimension. The finite difference stencils can be applied to these partial differentials to form a grid of spatial algebraic equations, examples of which will be introduced at a later stage. [3]

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial T}{\partial r}\right)+\frac{1}{r^2}\frac{\partial^2 T}{\partial \emptyset^2}+\frac{\partial^2 T}{\partial z^2}+\frac{1}{k}g(r,\emptyset,z)=\frac{1}{\alpha}\frac{\partial T}{\partial t} \quad (3.3\text{-}1)$$

3.4 Convective boundaries

A convective boundary describes the heat transfer between a solid and a fluid. Such a boundary would be necessary for determining steady state conditions of a battery which experiences cooling from its surrounding environment. It is possible to apply convective boundaries to a solid domain with the use of the fictitious node concept. If the domain for simulation is not expected to impact the external environmental temperature e.g. a small battery in a very large room, then an unchanging environmental temperature can be applied as a fictitious grid node which exists beyond the simulated domain. These can exist as nodes beyond the boundary terms to allow the application of the central difference formula right up to the grid boundaries to maintain a second order accurate solution.

Expressions (3.4-1) & (3.4-2) can be used to describe the convective heat transfer which occurs at the boundaries of a cylindrical cross section with 1D radial conduction, see Figure 5.5.2 for a visual understanding of the nomenclature expressed. Similar expressions can be described for Cartesian systems however are demonstrated for their desired application in this project. The terms $T_{\infty,a}$ & $T_{\infty,b}$ refer to the ambient temperature conditions surrounding the domain.

$$-k\frac{dT}{dr}+h_a T=h_a T_{\infty,a} \quad (3.4\text{-}1)$$

$$k\frac{dT}{dr}+h_b T=h_b T_{\infty,b} \quad (3.4\text{-}2)$$

These convective expressions may then be rewritten with the first derivative, central difference stencil shown previously (3.1-4). This provides solutions specific to the inner and outer boundary nodes for each iteration (3.4-3) & (3.4-4) where the coefficients β_o, β_N, γ_o, γ_M, G_o, G_M represent the relevant constant terms for both boundaries (3.4-5), (3.4-6) & (3.4-7).

$$2T_1-2\beta_o T_o+2\gamma_o+G_o=0 \quad (3.4\text{-}3)$$

$$2T_{N-1}-2\beta_N T_N+2\gamma_N+G_N=0 \quad (3.4\text{-}4)$$

$$\beta_o=1+\left[1-\frac{1}{2\left(\frac{a}{\delta}\right)}\right]\frac{\delta h_a}{k},\quad \beta_N=1+\left[1+\frac{1}{2\left[\left(\frac{a}{\delta}\right)+N\right]}\right]\frac{\delta h_b}{k} \quad (3.4\text{-}5)$$

$$\gamma_o=\left[1-\frac{1}{2\left(\frac{a}{\delta}\right)}\right]\frac{\delta}{k}\left(h_a T_{\infty,a}\right),\quad \gamma_M=\left[1+\frac{1}{2\left[\left(\frac{a}{\delta}\right)+N\right]}\right]\frac{\delta}{k}\left(h_b T_{\infty,b}\right) \quad (3.4\text{-}6)$$

$$G_o=\frac{\delta^2 g_o}{k},\qquad G_M=\frac{\delta^2 g_M}{k} \quad (3.4\text{-}7)$$

These solutions can solve the boundary terms at each iteration and would be used in conjunction with a heat conduction solution for the internal nodal values. It should be noted that for systems with 2D heat transfer applying adjacent convective boundaries requires the use of fictitious corner boundary nodes to apply the second order accurate central difference method. See Figure 4.4.2 where the fictitious nodes would be located at the boundary corners. [3]

3.5 Time dependency

There are many numerical schemes in place for solving time dependant conduction problems however the explicit form is of interest for its ease of implementation and comprehensive procedure. The general solution for the Cartesian Laplacian can be taken as shown in (3.5-1). The third spatial term z is not of interest as 2D heat transfer is desired, this term can be dropped and the finite difference stencils expressed previously (3.1-4), (3.1-5) can be applied to the derivative terms, forming the expression shown (3.5-3). This can be re-expressed in terms of a future temperature which is entirely dependent on grid temperatures from the previous time step.

$$\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}+\frac{\partial^2 T}{\partial z^2}+\frac{1}{k}g(x,y,z)=\frac{1}{\propto}\frac{\partial T}{\partial t} \tag{3.5-1}$$

$$\frac{T_{i-1,j}^{n}-2T_{i,j}^{n}+T_{i+1,j}^{n}}{\Delta x^2}+\frac{T_{i,j-1}^{n}-2T_{i,j}^{n}+T_{i,j-1}^{n}}{\Delta y^2}+\frac{1}{k}g(x,y,z)=\frac{1}{\propto}\frac{T_{i,j}^{n+1}-T_{i,j}^{n}}{\Delta t} \tag{3.5-2}$$

$$T_{i,j}^{n+1}=T_{i,j}^{n}+r_{sx}\left[\frac{T_{i-1,j}^{n}-2T_{i,j}^{n}+T_{i+1,j}^{n}}{\Delta x^2}\right]+r_{sy}\left[\frac{T_{i,j-1}^{n}-2T_{i,j}^{n}+T_{i,j-1}^{n}}{\Delta y^2}\right]+\frac{\Delta t\propto}{k}g(x,y,z) \tag{3.5-3}$$

The explicit method is significantly disadvantaged as the finite difference scheme implemented has what can be imagined as a rate of information transfer through the grid. For every iteration of the solution, information from the boundaries propagates through the grid however the pointwise nature of the system in combination with the applied stencil means that for fixed spatial steps; Δx , Δy and material diffusivity α, a maximum time-step Δt is imposed. The numerical solution becomes unstable if the time step exceeds the rate of information transfer through the grid. For every spatial dimension considered the stability criterion must be satisfied. Equation (3.5-3) contains two spatial stability ratios r_{sx}, r_{sy} these are expanded upon below (3.5-4). The summation of these spatial dimension stability ratios as a rule must be $\leq .5$ otherwise the system is vulnerable to become unstable and results in amplification of errors. This means that for each dimension added the allowable time step can become more and more restrictive as indicated with (3.5-6) when the spatial grids have the same step size.

$$r_{sx}=\frac{\alpha\Delta t}{(\Delta x)^2},\qquad r_{sy}=\frac{\alpha\Delta t}{(\Delta y)^2} \tag{3.5-4}$$

$$r_{sx}+r_{sy}\leq\frac{1}{2} \tag{3.5-5}$$

$$\Delta x=\Delta y\rightarrow r_s\leq\frac{1}{4} \tag{3.5-6}$$

The explicit method requires that spatial boundary conditions be known for all time steps and the solution can be initiated with an assumed initial grid state at time = 0. Figure 3.5.1 shows the 1D explicit finite difference stencil with time dependency. Each time iteration in the 1D case is dependent on three known previous grid temperatures, similarly the 2D case will depend on nine previous grid temperatures. This means that an iteration of a solved spatial grid represents one time step and as the

time is dependant only on the past the problem can be iterated through time to reach steady state conditions.

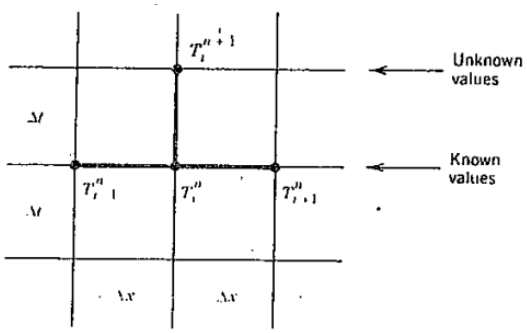

Figure 3.5.1: 1D finite difference stencil for explicit time dependency [3]

The significance of the restricted time step can now be better understood, as the smaller the time step, the greater the number of iterations and in turn computations required to reach a desired state in time. This means that the explicit solution is a very expensive method for simulating large periods in time. It is usually used for events which take place in very short time windows such as a simulation of an explosive charge. This means for simulation of a battery which may have a steady state condition of interest which takes minutes or hours to reach, trade-offs must be made between grid density & material diffusivity properties to achieve a steady state solution in a feasible time period. The explicit method features a truncation error of order O[$\Delta t, \Delta x^2, \Delta y^2$]. Alternatives such as the Alternating-direction-implicit (ADI) method would be better suited to solving the problems posed in this project however involve additional layers of complexity for their solution.

3.6 Internal heat generation

The general solution for the Cartesian co-ordinate Laplacian is shown (3.6-1), this contains a heat generation term $g(x,y,z)$ which can be spatially dependant or constant over the entire domain for simulation. A spatial heat generation grid can be initialised at the beginning of the program and used in the iterative solution; the grid would remain constant throughout the simulation. A similar term exists also in the cylindrical Laplacian representation (3.6-2). [3]

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} + \frac{1}{k} g(x,y,z) = \frac{1}{\propto} \frac{\partial T}{\partial t} \tag{3.6-1}$$

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial T}{\partial r}\right) + \frac{1}{r^2}\frac{\partial^2 T}{\partial \emptyset^2} + \frac{\partial^2 T}{\partial z^2} + \frac{1}{k} g(r,\emptyset,z) = \frac{1}{\alpha}\frac{\partial T}{\partial t} \tag{3.6-2}$$

In reality the internal heat generation in a battery is a by-product of chemical reactions taking place within, these would involve many complex and varied dependencies which would be impractical for simulation in the scope of this project.

3.7 Cluster Information

The project makes use of the Kelvin cluster available through Trinity College. This cluster features Intel Xeon hardware and InfiniBand interconnect. The specifications of the head node are provided in Figure 3.7.1 obtained using the l*scpu* command. It is understood that the cluster is made up of 100 identical Xeon nodes and so the description is relevant for both the head node, compute nodes and debug nodes utilised. These do not feature AVX but rather an older "sse4_2" instruction set with 16byte alignment. The *–xSSE4.2* compiler flag can be used to specify the use of vector extensions for improved vectorization. [5]

Figure 3.7.1: Kelvin cluster node specifications

4. Layout and Software Design

4.1 Overview

Table 9.1-1 contains a list of the appended code files related to this report. Individual files exist for running depending on the scenario for simulation. Computation of the various forms of the heat equation relies on a number of material properties and user defined variables. Some of these must be initialised by the user before compilation and vary depending on the desired; boundary conditions, material, grid shape, size and resolution, along with the number and type of dimensions under consideration. The primary project files are file 9.1-1, file 9.1-21, file 9.1-22 and are compiled and run with variables at run time however many others like boundary conditions remain hard-coded.

Each file holds a similar structure in that they take a user specified global grid and decompose this across a specified number of processors using functions from file 9.1-15. The decomposed grids are only ever allocated and held as local chunks on each processor. They are initialised to a starting state with functions from file 9.1-5, used to perform local solves of Laplace using finite difference methods and communicate relevant results with neighbours see file 9.1-2.

The Jacobi method is implemented by holding two sequential iterations of the local grids, using calculated from one update the other and visa versa. The two grids are compared to determine when convergence to a steady state solution has been obtained or a desired number of iterations have been exceeded (time-dependant problems rely purely on specified number of time steps), the convergence functions are available in file 9.1-2.

Finally the processors use MPI libraries to print their respective local grids to a global grid in a binary file using a function available in file 9.1-8. This file can then be converted to a .txt with a python script file 9.1-19 and imported to Mathlab software to generate the visuals presented later in this report. The Mathlab scripts can be found in file 9.1-23, file 9.1-24 and file 9.1-25.

4.2 2D Axial Processor Decomposition

With the quantity of nodes in each spatial dimension and number of processors specified by the user, the function *MPI_Rect_Axis_Decomp( )* can be called to divide processers amongst the available axis. It serves to optimise the distribution of workload as evenly as possible in each dimension by considering an optimal ratio of processors split over the nodes of the two axes (x, y or r, θ).

The significance of having even distribution of grid points in both axes will become clearer when the data exchange protocol is explained further in this section, however, to summarise if the ghost rows & columns which are to be communicated, store unequal quantities of data, the communication can become bottlenecked by the larger of the data exchanges.

There is an oversimplification at play here as the selection may not be the most optimized possibility in practice. Factors such as contiguous memory accesses may pay favour to one axis over another or the suggestion of utilizing a smaller number of processors than specified by the user is not considered.

For future implementations it should be noted that this function should not be used as there is an existing MPI function *MPI_Dims_create( )* which serves the same purpose and may include greater flexibility/optimization [6]. The operation is best understood visually with Figure 4.2.1, assume twelve processors are to be distributed over three different matrices. There are numerous ways this

could be decomposed for the shown grids; $1x12$, $2x6$, $3x4$, $4x3$, $6x2$ *or* $12x1$.. The function determines the ratio of grid nodes between both axes and forms a similar ratio of processors split between the two axes to best match. The benefit of this function is it allows running on quantities of processors which are not perfect squares and can better optimise rectangular grids.

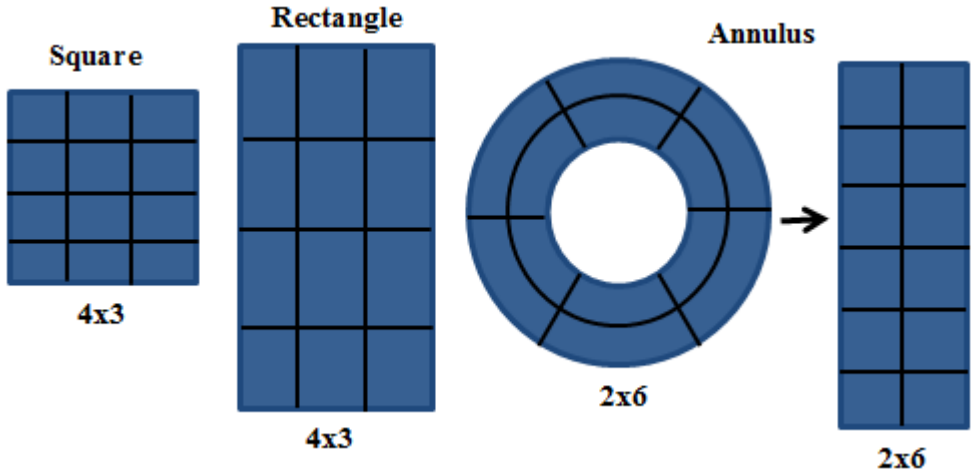

Figure 4.2.1: Decomposition of processors over axis for different shapes

MPI_Rect_Axis_Decomp( ) includes error checking for prime numbers of processors with exception to $1x1$, and so the $1x12, 12x1$ possibilities would not be allowed. It is assumed these would be handled by a separate one dimensional function although there is no reason this couldn't be included. Error checking is also present to ensure sufficient data exists for distribution across the quantity of processors specified, i.e. a square matrix with a total of four grid points couldn't be distributed among six processors.

4.3 2D Grid Decomposition

Next the function *MPI_Rec_Grid_Decomp2d( )* is called, taking the previously determined number of processors for each axis and decomposing the grid nodes in both axes amongst the processors. This is achieved by dividing the nodes in each axis by the respective number of processors, where not perfectly devisable, the remainder of nodes are obtained and an additional node is assigned to the first processors in a given axis until the number of the remainder is made up. This ensures the processors are load balanced and are responsible for a similar sized grid of nodes.

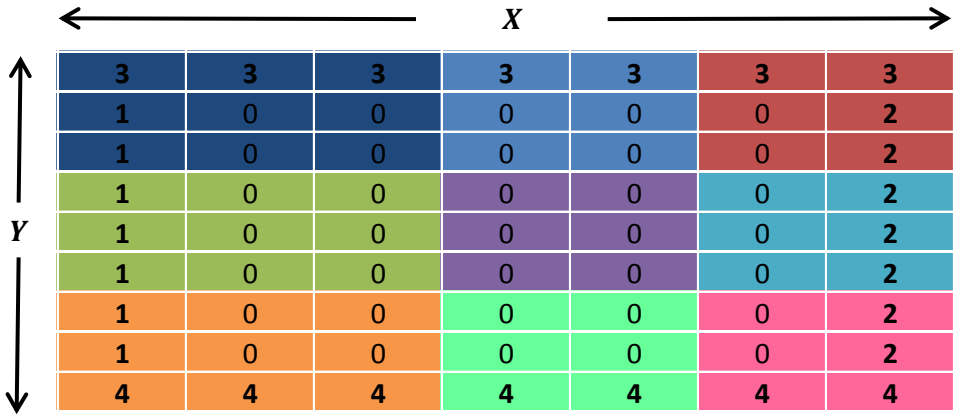

Figure 4.3.1: Decomposed 9x7 grid domain across nine processors

Figure 4.3.1 shows a $9x7$ grid domain to be decomposed. The nine processors allocated to the job have only one option for decomposition, three processors over each of the axes. While the rows split

evenly the columns do not and this results in the first column of processors obtaining an additional **x** dimension grid node. The processors are assigned their unique start and finish node values in each axis, representing their sub-section of a distributed grid. Continuing with the example shown, imagine the processors are numbered 0-8 and processor number four sits in the centre. This would have an x-axis start at the 4^{th} node from the left and end at 5^{th} while in its y-axis the start would be the 4^{th} node from the top and end with the 6^{th}.

An MPI library function *MPI_Cart_create( )* is called to create a communicator group for the specified number of dimensions and is immediately used in two calls of another library function *MPI_Cart_shift( ).* This obtains the immediate neighbouring processors of one another on processors on the global grid, this will be required for communicating data in each axis later. A significant item to note for *MPI_Cart_create( )* is the ability to specify periods, allowing the outermost edges of a grid to be looped back to the opposing ends. Illustrated in Figure 4.3.2, is an annular shape which will be considered and utilise this ability for simulation. The grid remains stored in the same rectangular form in memory, however abstractly, can be imagined in the form shown. Where $\boldsymbol{x}$ represents a **radial**-axis and $\boldsymbol{y}$ similarly the **tangential**-axis. The processors at the top can communicate to the processors at the bottom as if they were physically beside one another.

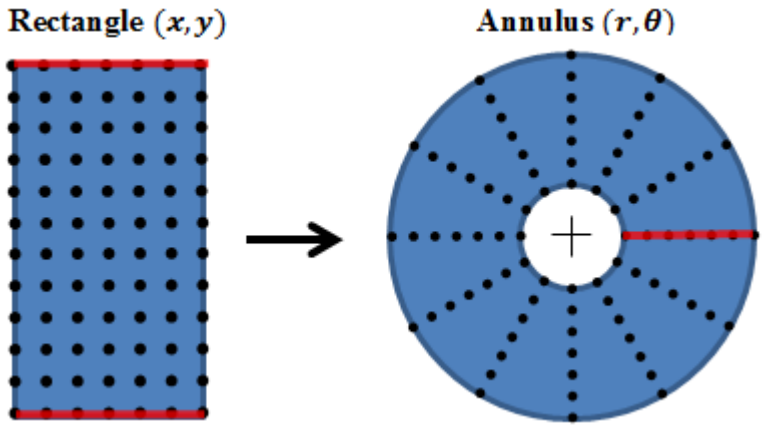

Figure 4.3.2: Annular representation stored as a periodic rectangular grid

4.4 Distributed Memory Allocation

alloc_2D_rect_matrix( ) is responsible for the dynamic allocation of memory for storage of grid data values at each node. To allow the data to be accessed and manipulated externally from the function using $Grid[y][x]$ notation, it is necessary to assign a contiguous chunk of memory and to pass the address of a double pointer into the function. Figure 4.4.1 is a representation of the memory allocation taking place, within the allocation function *A points to an array of Y-axis pointers. A new memory allocation is then made for a block of contiguous, local grid sized data and each of the Y array pointers are directed to point to the start of each X-axis data address. This means externally from the function a **A which had its address passed in can now point to any y, x grid location.

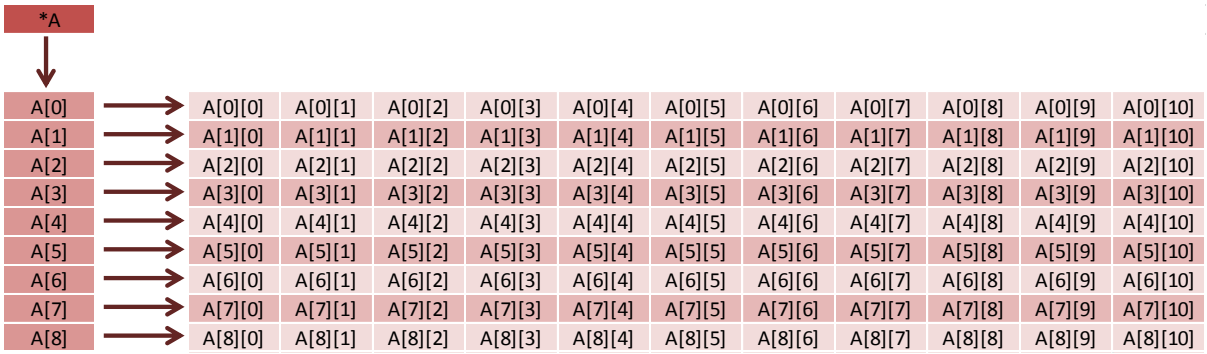

Figure 4.4.1: 2D Memory allocation using pointers to pointers

There are alternative methods for allocation of contiguous memory without the need for this additional Y-axis array of pointers and hence could be more memory efficient. This method was selected for its pointer notation as it is easier to comprehend at a quick glance over alternatives e.g. $A[j][i]\ vs\ A[j * x + i]$.

The requirement for data to be allocated dynamically stems from the codes parallel nature, before the grid is decomposed it is unknown what size local grid each processor will be responsible for. The reader may note throughout this body of work the Y-axis appears inverted to what one might expect. It is visualised as ascending order but in a downward direction. This is simply a preference as is how the author visualises memory stored in the computer; where M, N represent the row, columns respectively or the y, x axis.

For the purposes of scalability it was decided each processor would allocate memory only for the local grid it was assigned following decomposition. This enables the size of the grid solved to grow with the number of processors as each additional processor possesses its own memory cache. This is an example of a distributed-memory system [4], no one processor has access to the entire problem domain and so are only concerned with their local grids and neighbour ghost column/rows.

The code is run in a **SPMD** (Single Program Multiple Data) format, all functions and commands are written in globally understood terms which are processor specific. This is in contrast to the alternative **Master-Worker** scenario where an administrative master node might be responsible for holding the complete domain in memory and distributing the data to its worker processors for solution of delegated problems.

The issue that arises with the Master-Worker scenario is the need for communication of significant quantities of processor specific data, this is very time consuming and sequentializes the code as only one processor can receive data at a time and all others must sit idle during this time. The total domain size is then limited by the cache of a single processor and many processors resources are wasted.

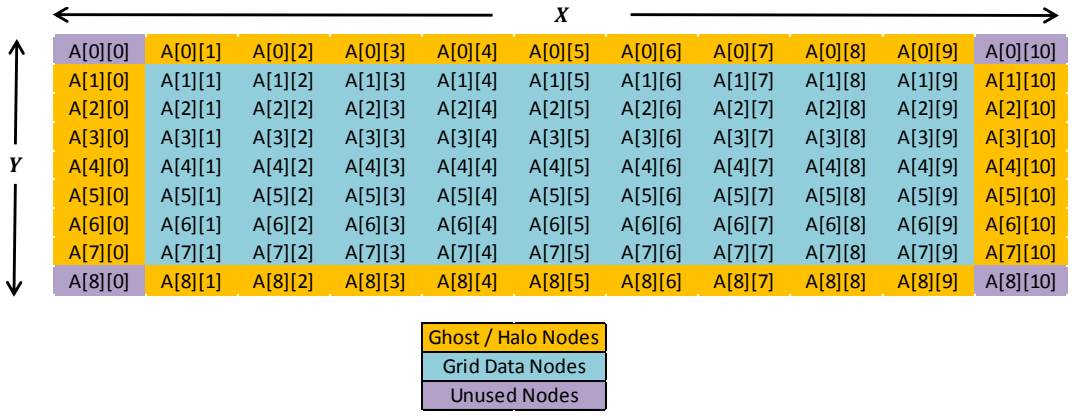

Figure 4.4.2: Local grid memory allocation with grid data ghost columns/rows

Figure 4.4.2 shows the local memory allocation on each node (dimensions would vary). The turquoise coloured memory locations are used to hold the processor specific local grid data designated during the grid decomposition. The yellow rows/columns are additional memory which must be added to provide Ghost or Halo storage and used to receive the outermost grid data from neighbouring processors. This will be discussed in more detail later; however these values are required for each processor to carry out their calculations for each iteration on their local grid. Lilac coloured columns exist at each of the corners and are excluded from the ghost communication; these are an artefact or by-product of the memory allocation and serve no purpose, never holding data (An exception to this rule can exist where adjacent convective boundaries condition exist).

4.5 Grid Value Initialisation

To illustrate how one can visualize the decomposition and memory structure of the distributed problem, Figure 4.5.1 represents the entire grid domain as a whole. This shows the decomposed local grids assigned to different processors as indicated by the variations in colour. To assign the initial conditions necessary for solving Jacobi all processors call the *Initialize_2D_Grid_Values ( )* function. This simply steps across all non-boundary nodes for each processor and assigns a prescribed value; zero in the example shown in Figure 4.5.1. The boundaries are set with a similar function *Initialize_2D_Grid_Boundaries( )* this takes a prescribed fixed value for each left, right, top and bottom boundary condition shown 1, 2, 3 and 4 respectively in the diagram.

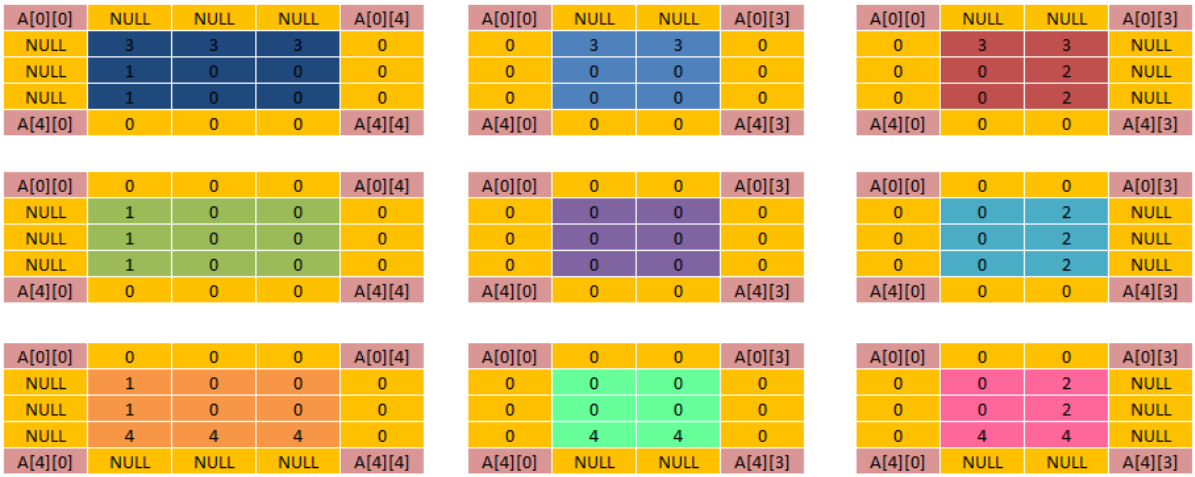

Figure 4.5.1: Distributed memory system across nine processors

These boundaries exist only on processors assigned to the boundaries of the global domain and are set using four if statements. These ordered statement calls can overlap at the edge boundaries, overwriting one another based on call order. For the example shown this results in the top left hand edge node becoming a three as opposed to a two. This is insignificant for some applications as the edge node serves no purpose in further calculations, however, in the case demonstrated previously in Figure 4.3.2, where a domain is periodic it becomes hugely significant. A pair of boundaries may no longer be required resulting in two extra rows or columns and the edge boundary must take the value of the remaining boundary. If attention is not paid here it may set to the value of a non-existent boundary.

This same grid more accurately exists as depicted in Figure 4.5.1, in its distributed form. Note that the memory addresses are local to each of the individual processors and so node A[0][0] is a representation in every processor. The processors with outer-most global grid boundary conditions contain one or more ghost row/columns which will not have any neighbouring processors; these boundaries serve no function and will be ignored if the domain is not periodic. Outside of the *MPI_Cart_shift( )* function discussed previously, neighbour processor indicators generated will be set to NULL to ensure no communication is attempted where neighbours do not exist. The value of NULL assigned to ghosts are just representative here however, and are not assigned any values in practice.

4.6 Jacobi Iteration

4.6.1 Iteration Loop

In this case the domain is solved using the Jacobi method, an iterative solver, so it is necessary to have a for loop which continues until either some convergence criteria is met or a maximum number of iterations is exceeded and it is no longer desired to continue. Within this for loop the functions *Exchange_Data_2D( )* and *Sweep_Solve_2D( )* are called in an alternating pattern twice before the loop repeats. The reason for this is twofold; Unlike with Gauss-Seidel iteration Jacobi retains a complete copy of the most recent nodal positions without overwriting the values as it solves the grid. In this case it is achieved by creating and allocating an identical second grid on each processor and alternating between the two, placing the results of one in another and vice versa.

The second use is capturing two sequential iterations of the grid in memory which can be compared to determine convergence and hence offer an exit strategy from the iterative loop.

4.6.2 Exchanging Processor Data

The *Exchange_Data_2D ( )* function is used to send the first and last rows/columns of a processors grid data to their neighbouring processors ghost and similarly receive neighbouring grid data into their own ghost rows and columns as illustrated in Figure 4.6.1 with a column exchange. Processor 1 sends to the right and receives from the right; similarly Processor 2 sends and receives to and from processor 1 on its left. In the two dimensional application this exchange occurs in four directions, within the function are eight MPI library function calls between *MPI_Isend( )* & *MPI_Irecv()* for each direction.

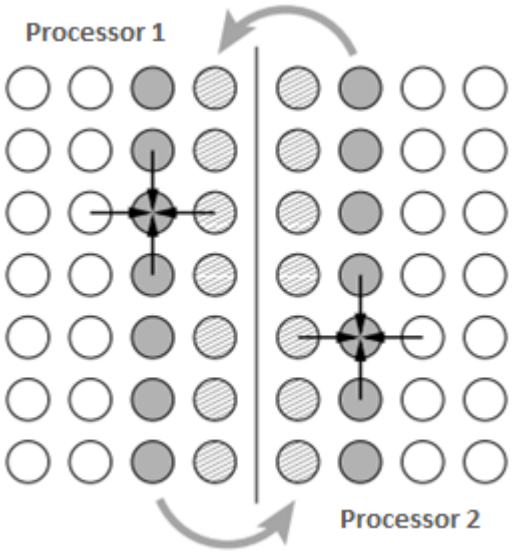

Figure 4.6.1 : MPI column data exchange between two neighbouring nodes from data to ghosts [6]

The MPI library provides a variety of similar send/receive functions however these have been used specifically because they are an example of non-blocking communication. This avoids the possibility of a deadlock occurring, which is where all processors attempt to send at the same time and must wait for the message to successfully be received before continuing. In the case where this communication is blocking each processors own send prevents them from receiving their neighbours sent messages and so no processors message is ever communicated, there is a freezes or deadlock.

MPI_Isend( ) & *MPI_Irecv( )* allow the processors to continue with other operations while a communication takes place, this can be to communicate with their other neighbour processors. The avoidance of the blocking waiting periods enables much more efficient communication which would otherwise become sequentialized and bottleneck the performance. There is a limit to the work that can done by the processors while this data exchange is occurring, the exchange function cant end until all processors have completed communication as to do otherwise would mean performing a calculation with data that may or may not have been successfully exchanged. *MPI_Waitall( )*, another inbuilt MPI function prevents processors from continuing beyond this point until all communications in each direction have been completed. [4]

When sending and receiving columns as illustrated in Figure 4.6.1 a new datatype is necessary as the data will not be contiguous in the Y direction with the method used to allocate memory described

previously. To access each element of data in the Y dimension an entire row in the X dimension data must be skipped over. To enable the sending of this non-contiguous data a new MPI_Datatype was created to represent a column as a vector of data. This datatype is required for use in the *Exchange_Data_2D( )* function however to improve performance and prevent repeated allocation and freeing of the datatype it has been declared external to the function call and prior to the iteration f*or( )* loop to allow a single allocation and free call.

4.6.3 Applying Finite Difference Calculation

Sweep_Solve_2D( ) performs the main calculation, sweeping through the local processor grids it applies the finite difference formula derived for the specific problem in question at every grid node to solving for an updated nodal value. The formula solved will need to be swapped out or altered for different simulation circumstances. These could include time dependent problems, a 2D steady state problem on an annulus shape with polar co-ordinates, internal heat generation or for use with convective boundary conditions. The implementation of convective boundary conditions introduces some additional complexity to this function as with each sweep the convective boundary nodes must also be updated where they would otherwise remain constant for fixed boundary problems. This can be achieved with an *if( )* statement to check first whether a processor exists at a global grid boundary. If the condition is met a *for( )* loop applies a convective boundary specific function to be solved at these nodes.

4.6.4 Convergence Check

As the two matrices approach a solution the rate of change or difference between them should tend to zero. The criteria for measuring non-transient convergence to steady state is frequently done by solving the one norm $||A||_1$, two norm $||A||_2$, Euclidian norm $||A||_E$ or infinity norm$||A||_\infty$, and comparing these against some threshold value.

Some of these may be easily calculated for square matrices but become slightly more involved for rectangular dimensions; the solution becomes more computationally intense as it can require performing matrix operations on large grids, requiring memory allocation to store large matrices or vectors.

For some matrices the one norm can be generalised simply as the maximum of one–norms for all the column vectors of A. $||A||_1 = \max\{||A_i||: A_i \; is \; a \; column \; vector \; of \; A\}$ and the infinity norm could then be the maximum of the one-norms over all of the row vectors in A $||A||_\infty = \max\{||A_i||: A_i \; is \; a \; row \; vector \; of \; A\}$ [7].

Determining the largest value of a vector would involve checking each vector position against a current maximum with an *if( )* statement to overwrite this value when true. *If( )* statements prevent vectorisation in loops so it might be expected to achieve a slower solution, depending on how frequently convergence is checked this may become a problem. The other issue is that the vectors (row/columns) needed for the norms are distributed and would require additional inter-processor communication for each vector.

The selected method usually depends on the user. This encouraged the development of the convergence function *Calc_grid_diff_pointwise( )* a convergence checking function which avoids the calculation of any matrix norms and instead aims to return a value which indicates whether every element in the matrix is below a user requested tolerance value.

It measures the difference between grid nodes in two sequential iterations and squares the difference to ensure a positive value; this result is then divided by the squared tolerance value prescribed by the user. If the difference is greater than or equal to tolerance, a value ≥ 1.0 is obtained, similarly once the difference is below tolerance the result will be < 1.0. The maths library function *floor( )* can then be used to round to the nearest integer value removing the fractional part of the number. This instead means the value would be exactly zero if the solution was less than tolerance or non-zero otherwise.

By summing the results of every grid node position an indicator can be returned which is zero in the event where all nodes are within the tolerance or some non-zero term indicating the criteria is not yet met. One issue however with this solution is that the *floor( )* is understood to be quite computationally expensive and would be required for use at every grid node. The solution implemented is instead to down-cast the double to an integer before it is summed which has a similar effect in removing fractional numbers.

The process of down casting is susceptible to issues with overflow; double precision numbers can represent much larger values with 8bytes of storage than their integer, 4byte counterparts [8]. Values will be undefined if the double being cast represents numbers exceeding that allowed by the integer. These undefined values may even become negative representations however because the result is needed only to act as an indicator the accuracy of the numerical value becomes irrelevant. The result will only become undefined where the matrix does not satisfy the criteria and will be well defined when the matrix does meet the expected tolerance.

The grid is distributed which means each processor is required to perform a local summation for the convergence criteria and share their results to all other processors using the blocking collective communication *MPI_Allreduce( ).* This enables each processor to form a result for the summation of convergence criteria over the entire grid. A simple if statement is then used to break the Iteration loop where the convergence criteria satisfies.

It should be noted that such convergence criteria is not used in time dependent simulations as the user specified time step would influence the difference values obtained between iterations. This means generally transient simulations rely solely on specified time duration for simulations to know when to stop.

4.7 Parallel Save to File

Once the simulation has converged or the maximum number of iterations have been exceeded *Parallel_Print_Double_To_File_wBoundary( )* is called to amalgamate the global grid in a single file from each of the local grids held on the processors, similar to that shown in Figure 4.3.1. Each processor opens the same file of a user specified name with the function MPI_File_open( ). This is a special MPI-IO function which provides each its own designated file pointer and allows parallel transfer of data as processors can write to file simultaneously.

This type of parallel print function is a necessity for the distributed type system implemented. A Manager-worker printing scenario would involve each processor sending its entire grid to an administrative node which would be responsible for printing the whole grid. This would significantly serialize the code as each processor would be required to wait its turn to send to the manager processor, but also for the manager to print the data to file. There may be complications to consider regarding memory allocation with any minor variations in grid sizes of each processor too.

Although the function is only called once at the end of the steady state solution, such serialization on a problem with a significant number of processors would become very wasteful as all but one remain idle during this final communication phase. It may also be desired for multiple prints when performing transient solutions and hence this parallel printing capability becomes a necessity. This process is not without its own caveats; the output file resulting from this function is not immediately user readable and requires a further process to interpret/reformat the data. This challenge can be overcome by reformatting with serial code, run on an individual processor post simulation.

Implementation of this parallel write function is non-trivial, the functioning code itself becomes difficult to follow and was the most time consuming portion of the project to implement successfully. As mentioned previously the output format is not easily user readable making troubleshooting more challenging.

In order for each processor to navigate the file they form personalised datatypes with *MPI_Type_contiguous( )* which describe how the data is to be contiguously printed to file despite the need to skip over other processors data positions. This is necessary to print more than one row to file. Another datatype is generated to describe how the data can be contiguously read from the processors local grid while avoiding the undesired ghost column data.

The function *MPI_File_set_view( )* is then used to create a sort of writeable window in the save file where the processor is allowed to transfer data. It hides the information of the other processors and returns a pointer to the initial position in the file where the data is to be saved. Each processor naturally will have a different offset in the save file and the file view prevents each overwriting one another's data.

Finally the processors use the function *MPI_File_write ( )* to transfer the data from their local grids to the save file. The ghost rows of the local grids are avoided by providing the function a pointer to the initial grid data position and restricting the number of the contiguous datatype sends to the size of the y-axis, avoids the bottom ghost row.

4.8 Binary to Decimal File Conversion

The file *Binary_to_decimal_txt.py* file 9.1-19 was written to handle conversion of the output MPI-I/O binary files generated during simulation to a decimal string representation in a ".txt" file. The binary format is not easily human readable, particularly where double precision numbers are involved, so this conversion is necessary for troubleshooting the main Jacobi solver and for using results with other visualisation tools. This file can be altered to suit either *int* or *double* precision values however for these simulations double precision is desired. The binary output files store the double precision data using the Big-Endian binary format on the chuck cluster and the executable is written with this in mind. An example of this format is provided in Table 4.8-1 for the double precision representation of the number one.

The file can be altered to specify the digits of decimal precision to be printed to file. It is not understood whether the Big-Endian format is consistent with MPI or if it is based on the native device with which the *MPI_File_write( )* function is performed. This poses a possible source of error which may need further investigation. Referring back to Figure 3.7.1, the native format of the hardware these functions were tested on should have Little Endian which would suggest MPI uses a fixed format for the sake of portability.

Table 4.8-1: Numerical Representation of different datatype formats

Datatype formats	Numerical Representations
Big Endian Binary	00111111 11110000 00000000 00000000 00000000 00000000 00000000 00000000
IEE 64-bit breakdown	sign Exponent Mantissa 0 01111111111 0000 00000000 00000000 00000000 00000000 00000000 00000000
Decimal	1.0000000000000000

Because this code is specifically expected to be run on a single processor and has no need for parallel processing or performance considerations the conversion file was written in Python, a more functional, higher level language. The executable takes three input variables; the number of grid columns ***M***, the number of grid rows row ***N*** and the ***filename*** to be converted. The executable will overwrite any existing file if one exists with the same output filename.

The code takes the input filename appending "_decimal.txt" and creating a new .txt file from it. It then opens the original .o file and reads in eight bytes of information at a time (for double precision), converting to binary and saving the new binary representation to the .txt output file before the next step. After each element is added a space delineator is placed to prevent the elements merging to form one number. This makes the grid easier to read and use with other software's later. For each new line of the matrix read in, one is added to the output file and so the grid structure is maintained, this is made possible by counting to the input M value and similarly the process is complete when all M,N values have been accounted for.

5. Testing

5.1 Overview

It would not be practical to cover all of the testing performed throughout the production of the code as such, a number of items have been prioritized based on their significance and presented in the following subsections. It may be worth noting briefly however that a number of files exist in the code base that were used to stage testing of functions and their compatibility when interacting together.

The decomposition functions were tested across varying grid sizes and axis dimensions to ensure each processor was designated unique sub-sections of the global grid. They were also used to verify error handlers returned correct failure statements/killed execution when run with invalid parameters see file 9.1-13 and file 9.1-14.

The two dimensional memory allocation function may appear straightforward when presented visually in this report however this was an item that required its own test file for troubleshooting. The use of a programming tool called Valgrind was also implemented to ensure that memory was successfully free'd after use and no memory leaks would occur, file 9.1-7. This test file made use of a number of other small functions written and used for troubleshooting such as a serial grid print function and functions to initialize grid data which can be found in file 9.1-5.

Extensive testing was performed in both the *Parallel_Write_to_file( )* function and the production of a binary to decimal file converter. These two items were written iteratively; first for *int* types which are more easily read from a binary file and then for *double* types tested using file 9.1-12. Because these were test files they were not maintained up to date with the primary code base beyond fulfillment of their initial purpose and so, may no longer function as intended.

Two primary files not covered in the results section exist, one for simulation with transient heat conduction file 9.1-20**Error! Reference source not found.** and another with convective boundaries file 9.1-22. Although time was spent on implementation and troubleshooting of these methods no usable results could be obtained for inclusion and the implementations remain incomplete. The convective boundary problem causes results tend to infinity, the functions implemented were then checked with some hand calculations and found to feature a similar phenomenon. The time dependent implementation was abandoned due to time constraints.

5.2 Undefined behaviour of down casting

Figure 5.2.1 presents a short test conducted to show the overflow phenomenon of undefined behaviour arising from casting from ***doubles*** to ***ints,*** mentioned previously when discussing the structure of the code, in particular the convergence criteria. This rough test acts as a proof of concept to examine the undefined behaviour over a range of values. A loop increments the value of a ***double*** precision number in powers of base 2. This is then cast to an ***int*** and printed to the terminal shown on the right of the image. The value of the incremented numbers rapidly grow beyond the integer representation capabilities of 4bytes, after which they are seen to stop growing and maintain an undefined negative representation shown in the bottom right corner.

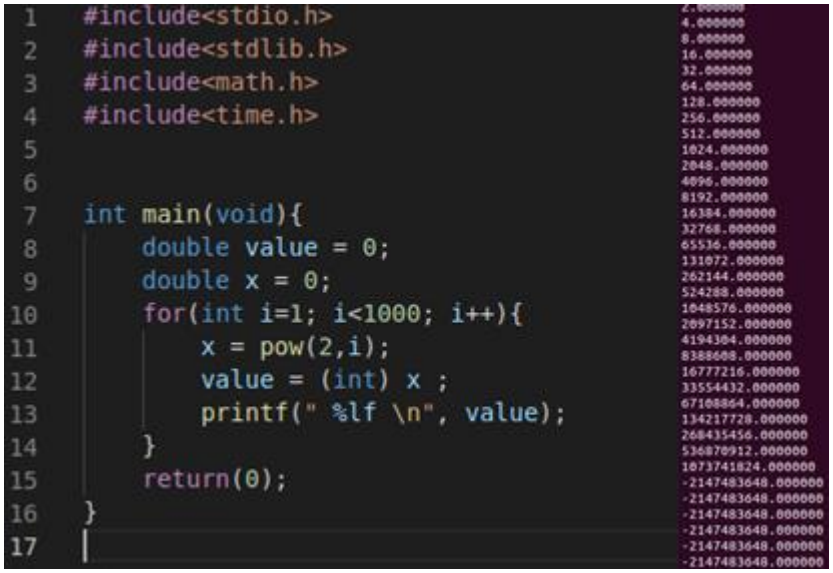

Figure 5.2.1: Test of undefined behaviour when casting double to int

Although this number is undefined and negative it will not cause the convergence criteria to be met (*convergence == 0*). That is, provided the criteria for convergence is not specified as $\leq$ zero and rather is specified as exactly equivalent to zero.

5.3 Computational cost of convergence function options

It was believed *if( )* statements and *floor( )* functions were potentially expensive operations which would prevent vectorisation. A simple test code was devised to compare the different computational expenses between down casting, using the *Math.h* library *floor( )* function and performing *if( )* statement checks required for a norm calculation. The code is shown in Figure 5.3.1, and involves simply three timed *for( )* loops over a large iteration number.

The first loop examines the time taken for down casting; this is done by performing an arbitrary calculation to obtain a ***double*** precision number "**x**"**.** The number is next cast from double to ***int*** before the result is added to "value". By accumulating the result in "value" it can be compared with the other loop results, however the main purpose for the accumulated value and also the arbitrary calculation involving the changing loop iteration number, is actually to avoid certain compiler optimizations. If the compiler believes unnecessary work is being performed i.e. calculations where the results are unused or where the result is unchanging and predictable it may otherwise optimize out this work and provide a false completion time. This work sits within a large loop in order to consume sufficient processing time to obtain a reliable result free from other computing noise that could cause time to fluctuate (i.e. context switching).

The second, loop is identical with the exception it uses the *floor( )* function in place of the cast "(int)". The third loop involves an interrupting if statement, the purpose in this case was to simulate the work that would be required to calculate a matrix norm. This would require a check on every node in a matrix to determine the largest value of a matrix, comparing a current largest with each node and overwriting the current largest each time a greater value was obtained. Because of this the loop includes an *if*( ) statement that performs a similar check on the "sum" value before overwriting a "max".

Each loop requires similar arbitrary tasks to obtain a "value" result and print to screen as without the requirement to do something optimization flags would cut out the *for( )* loops and *if( )* statements altogether.

```
18  int main(void){
19      double sum1 = 0;
20      double sum2 = 0;
21      double sum3 = 0;
22      double value1 =0;
23      double value2 = 0;
24      double value3 = 0;
25
26      double x = 0;
27      clock_t time_1 = clock();
28
29      for(int i=1; i<1000000000; i++){    // downcasting
30          x = (i +.15)/2;
31          sum1 = (int) x;
32          value1 += sum1;
33      }
34
35      clock_t time_2 = clock();
36      for(int i=1; i<1000000000; i++){    // floor() function
37          x = (i+.15)/2;
38          sum2 = floor(x);
39          value2 +=sum2;
40      }
41
42      clock_t time_3= clock();
43      double max = 0;
44      for(int i=1; i<1000000000; i++){    // interupting if() statment
45          x = i*i;
46          sum3 = x;
47          value3 +=sum3;
48
49          if( sum3 > max){
50              max = sum3+1;
51          }
52      }
53      clock_t time_4= clock();
54
55      double first_loop = time_2-time_1;
56      double second_loop = time_3-time_2;
57      double third_loop = time_4-time_3;
58      printf("first:%lf second:%lf third:%lf \n", first_loop, second_loop, third_loop);
59      printf("value1= %lf, value2=%lf , value3=%lf, max: %lf \n", value1, value2, value3, max);
60      return(0);
61  }
```

Figure 5.3.1: Test of computational expense of different convergence function options

Two sets of results were obtained, the first using the -O0 compiler flag to prevent optimization. The second with maximum optimization using the –O3 flag. The results are shown in Table 5.3-1 and indicate that with no optimisation in place, all loops perform their relevant tasks in a very similar time. The third loop shows a fractional improvement however, when run numerous times again minor fluctuations in results make this seem insignificant. With full optimization enabled all loops experienced a noticeable improvement in run time which might indicate none prevented vectorisation from occurring as was suspected previously. It is shown in loop 1 that the use of down casting is

more advantageous in execution time when compared with the *floor( )* function. Notably the best performance is seen in the third loop with the use of the *if( )* statement and although some fluctuation occurs with re-runs this performance difference is consistent.

Table 5.3-1: Computational Expense results for different convergence function options

	Loop 1	Loop 2	Loop 3
-O0 Time (s):	2862058.	2831172.	2764546.
-O3Time (s):	927974.	1798370.	753441.

The arbitrary output values of loops 1 and 2 matched one another in both cases and all output values obtained matched their respective values in both execution cases. This means there was no impact on the numerical values obtained with the higher optimisations applied. From this test the use of down casting is validated as a more efficient means of removing the fractional part of a number than its *floor( )* function counterpart. Although loop 3 (*if( )* statement) outperformed loop 1 (down cast) this run does not consider the additional communications which would need to take place for the Norm calculation and would incur execution cost which may further validate the selected mechanism for calculating convergence.

5.4 Square Grid Fixed Boundary Steady State

5.4.1 Simulation Conditions

To verify the output of the written program it is necessary to compare the simulated numerical solution with that of an analytical one of identical circumstances and environmental variables. A suitable analytical solution was first obtained and the numerical simulations run using these same input variables. Figure 5.4.1 shows the problem that was to be solved in both cases, where a, b are the dimensions in meters of a square block and four fixed boundary conditions are provided at each edge. The reader should note the form the x, y axis takes in this case, which will be flipped on the x axis when represented/replicated in the code. Three of the boundaries are identical and the final boundary is variable with x. This should result in a maximum boundary temperature occurring at $T\left(\frac{a}{2}, b\right)$ which reduces to T_1 at $T(0, b)$ & $T(a, b)$.

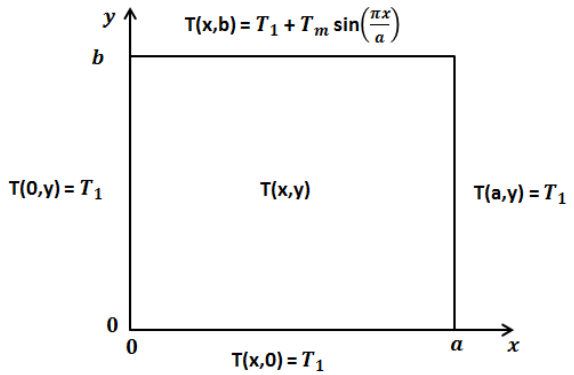

Figure 5.4.1: 2D Square fixed boundary SS problem nomenclature

This was performed on a square grid where $a = b = 1m$, and boundary temperatures were $T_1 = 20℃$ and $T_m = 100℃$. The simulation takes place with no internal heat generation and to steady state meaning there are no material property dependencies.

5.4.2 Analytical Solution Procedure

An Analytical solution can be derived from the general solution for the Laplacian shown in (5.4-1) for the Cartesian co-ordinate system for a cube shaped object. In this case a two dimensional steady state solution is desired so the time dependant and z dimensions can be dropped to give formula (5.4-2).

$$\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}+\frac{\partial^2 T}{\partial z^2}+\frac{1}{k}g=\frac{\partial T}{\partial t}\frac{1}{\propto} \tag{5.4-1}$$

$$\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}=0 \tag{5.4-2}$$

The separation of variables technique can be applied to rewrite these partial differentials as two independent differential equations with respect to a parameter lambda shown (5.4-8).

$$T(x,y)=v(x)w(y) \tag{5.4-3}$$

$$\frac{\partial^2 T(x,y)}{\partial x^2}=\frac{\partial^2 v(x)}{\partial x^2}w(y)=v_{xx}w \tag{5.4-4}$$

$$\frac{\partial^2 T(x,y)}{\partial y^2}=v(x)\frac{\partial^2 w(y)}{\partial y^2}=vw_{yy} \tag{5.4-5}$$

$$v_{xx}w+vw_{yy}=0 \tag{5.4-6}$$

$$v_{xx}w=-vw_{yy} \tag{5.4-7}$$

$$\frac{v_{xx}}{v}=-\frac{w_{yy}}{w}=\lambda \tag{5.4-8}$$

These differential equations can then be integrated for the possible conditions of lambda to generate three possible expressions of each of the separated functions. The possibilities for lambda will be $\lambda=0,\ \lambda<0,\ \lambda>0$. Where lambda is equal to zero both expressions can be simply integrated to form linear functions shown in equation (5.4-10).

$$\begin{cases} v_{xx}-\lambda v=0 \\ w_{yy}+\lambda w=0 \end{cases} \quad \begin{cases} \lambda=0 \\ \lambda\neq 0 \end{cases} \tag{5.4-9}$$

$$\lambda=0 \ \begin{cases} v(x)=c_1x+c_2 \\ w(y)=c_3y+c_4 \end{cases} \tag{5.4-10}$$

Where lambda is a non-zero, two possible expressions will exist which can be expressed in terms of trigonometric functions (5.4-17) and (5.4-20).
$\lambda\neq 0$:

$$w(y)=e^{\sigma y}\rightarrow w_y=\sigma e^{\sigma y}\rightarrow w_{yy}=\sigma^2 e^{\sigma y} \tag{5.4-11}$$

$$\sigma^2 e^{\sigma y}+\lambda e^{\sigma y}=0\ \rightarrow\ \sigma^2+\lambda=0\rightarrow\ \sigma=\pm\sqrt{-\lambda} \tag{5.4-12}$$

$$w(y)=\ c_3e^{\sqrt{-\lambda}y}+\ c_4e^{-\sqrt{-\lambda}y} \tag{5.4-13}$$

$$\begin{cases} \lambda < 0, & \lambda = -\omega^2 \quad w(y) = c_3 e^{\omega y} + c_4 e^{-\omega y} \\ \lambda > 0, & -1 = i^2 \quad w(y) = c_3 e^{i\sqrt{\lambda} y} + c_4 e^{-i\sqrt{\lambda} y} \end{cases} \tag{5.4-14}$$

$\lambda < 0$:

$$w(y) = c_3 e^{\omega y} + c_4 e^{-\omega y} \rightarrow c_3(\cosh(\omega y) + \sinh(\omega y)) + c_4(\cosh(\omega y) - \sinh(\omega y)) \tag{5.4-15}$$

$$State: \quad C = c_3 + c_4, \qquad D = c_3 - c_4 \tag{5.4-16}$$

$$w(y) = C\cosh(\omega y) + D\sinh(\omega y) \tag{5.4-17}$$

$\lambda > 0$:

$$w(y) = c_3 e^{i\sqrt{\lambda} y} + c_4 e^{-i\sqrt{\lambda} y} \rightarrow c_3(\cosh(\sqrt{\lambda} y) + i\sin(\sqrt{\lambda} y)) + c_4(\cosh(\sqrt{\lambda} y) - i\sin(\sqrt{\lambda} y)) \tag{5.4-18}$$

$$State: C = c_3 + c_4, \qquad D = i(c_3 - c_4) \tag{5.4-19}$$

$$w(y) = C\cosh(\omega y) + D\sin(\omega y) \tag{5.4-20}$$

Performing a similar approach the two other forms for $v(x)$ finally gives the three possible forms each of the terms (5.4-21) and (5.4-22).

$$w(y) = \begin{cases} C_1\cosh(\omega y) + D_1\sinh(\omega y), & \lambda < 0 \\ C_2 y + D_2, & \lambda = 0 \\ C_3\cosh(\omega y) + D_3\sin(\omega y), & \lambda > 0 \end{cases} \tag{5.4-21}$$

$$v(x) = \begin{cases} A_1\cos(\omega x) + B_1\sin(\omega x), & \lambda < 0 \\ A_2 x + B_2, & \lambda = 0 \\ A_3\cosh(\omega x) + B_3\sinh(\omega x), & \lambda > 0 \end{cases} \tag{5.4-22}$$

[9] [3]

The boundary conditions unique to the problem being solved can then be applied to both $v(x),\ w(y)$. These boundary conditions and the knowledge that the equations must satisfy for all values $x = 0 \rightarrow a$ *and* $y = 0 \rightarrow b$ provide the ability to narrow down and discard of incompatible functions until only one possibility remains for each $v(x),\ w(y)$. Substituting these equations back into the original form shown in (5.4-3) will give the final analytical solution, in our case provided (5.4-24).

$$v(0) = T_1, \qquad v(a) = T_1, \qquad w(0) = T_1, \qquad w(b) = T_1 + T_m\left(\frac{\pi x}{a}\right), \tag{5.4-23}$$

$$T(x,y) = T_1 + T_m \frac{\sinh\left(\frac{\pi y}{a}\right)}{\sinh\left(\frac{\pi b}{a}\right)} sin\left(\frac{\pi x}{a}\right) \tag{5.4-24}$$

[10] [9]

This is significant because it means a grid of analytical values can be generated for comparison with those sourced numerically. Mathlab was then used to apply this analytical solution to generate a 500x500 meshed grid and plot visual representations of the heat distribution over the 2D surface.

5.4.3 Interpretation of Analytical Results

Figure 5.4.2 shows the 3D surface plot of the temperature distribution obtained from running the analytical calculation. Three of the boundaries possess equivalent temperatures, $T_1 = 0$℃ as expected with a fourth boundary varying with the x dimension, exhibiting a maximum temperature of $T_m =$

100℃. A similar Figure 5.4.3 shows the same data however displayed as a contour plot looking top down at the 2D shape.

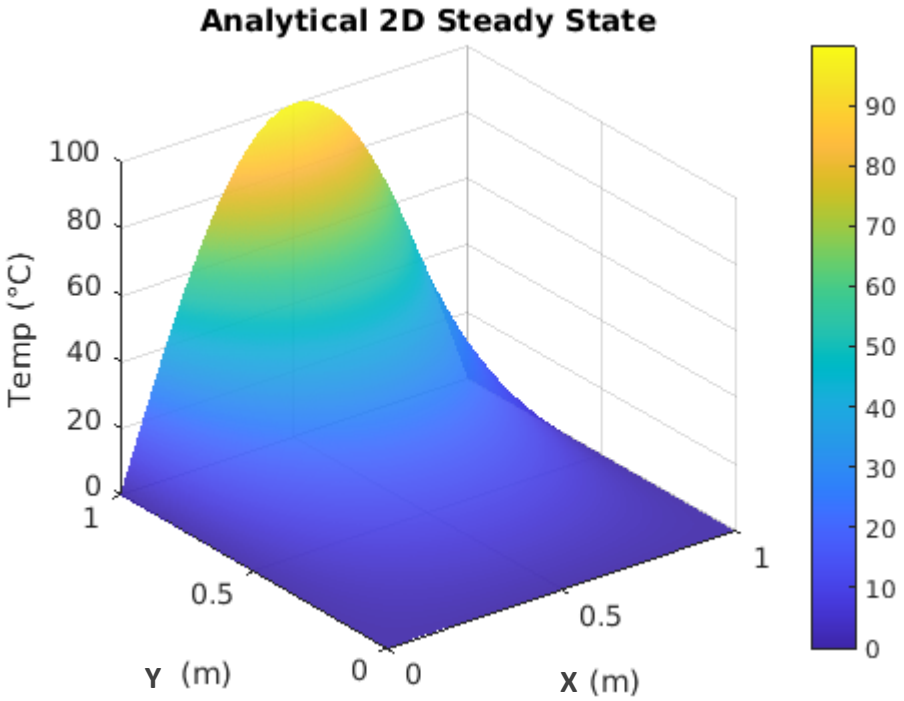

Figure 5.4.2: Surface plot of analytical solution of square grid fixed boundary SS

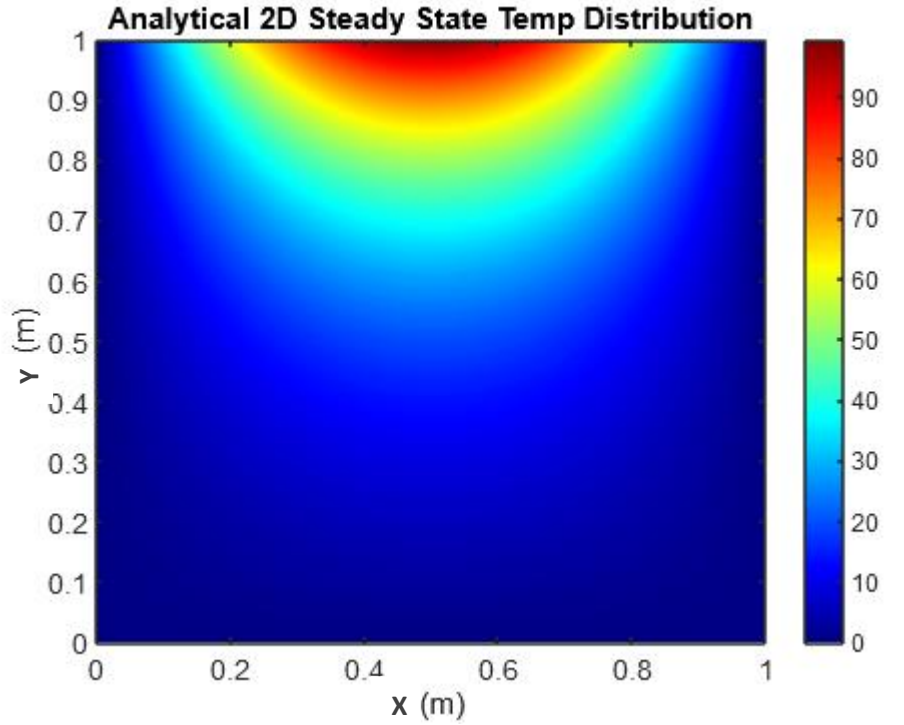

Figure 5.4.3: Contour plot of analytical solution of square grid fixed boundary SS

5.4.4 Numerical Solution Procedure

An expression for calculation of the iterative numerical solution can be obtained from the general solution of the Laplacian shown in (5.4-25) for the Cartesian co-ordinate system for a rectangular cuboid shaped object. In this case a two dimensional the steady state solution is desired so the time dependant and z dimensions can be dropped to leave the 2D steady state formula.

$$\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}+\frac{\partial^2 T}{\partial z^2}+\frac{1}{k}g=\frac{1}{\propto}\frac{\partial T}{\partial t} \tag{5.4-25}$$

$$\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}=f(x,y) \tag{5.4-26}$$

The second order accurate central difference technique is used to provide a three point expression to replace the second order derivative terms.

$$\begin{aligned} f_i'' &= \frac{1}{h_i^2}(f_{i-1}-2f_i+f_{i+1}) \\ f_j'' &= \frac{1}{h_j^2}(f_{j-1}-2f_j+f_{j+1}) \end{aligned} \tag{5.4-27}$$

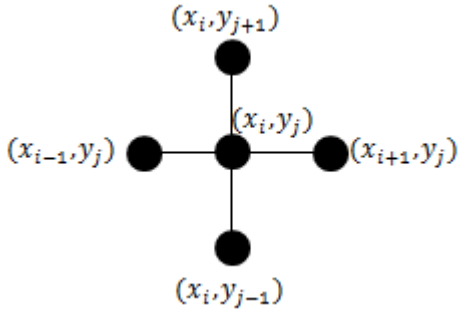

Figure 5.4.4: Five Point Finite Difference Stencil

When applied to both axes a common central node exists forming the five point stencil in Figure 5.4.4. The formula can be represented as (5.4-28).

$$f(x,y)=\frac{1}{h_i^2}\left(f_{i-1,j}-2f_{i,j}+f_{i+1,j}\right)+\frac{1}{h_j^2}(f_{i,j-1}-2f_{i,j}+f_{i,j+1}) \tag{5.4-28}$$

If the grid step sizes are made equal $\Delta x = \Delta y$, this can be reduced to (5.4-30).

$$f(x,y)=\frac{1}{h^2}\left(f_{i-1,j}+f_{i,j-1}-4f_{i,j}+f_{i+1,j}+f_{i,j+1}\right) \tag{5.4-29}$$

$$f_{i,j}=.25\left(f_{i-1,j}+f_{i,j-1}+f_{i+1,j}+f_{i,j+1}\right)-h^2 f(x,y) \tag{5.4-30}$$

This function can then be used to iteratively solve for a future value of $f_{i,j}$ using previous iteration values as input (Jacobi Iteration) and in the case of the very first calculation using assumed input values. This equation was then used to solve for internal (non-boundary) grid points in the *Sweep_Solve_2D( )* function available within file 9.1-2.

The problem boundary conditions were written into a designated file 9.1-1 with the specified square grid dimensions. This was then compiled using –xSSE4.2 and -03 optimization flags and executed for a 500x500 grid with a maximum iteration of 500,000 (high enough to ensure convergence occurred). The problem was run on the Kelvin cluster across 4 processors, although it will be shown in a stability testing section that the number of processors has no impact on the numerical values obtained. The program was run four separate times, with varying tolerance, a user input convergence requirement. This was reduced by a factor of ten each time in order to show the effect when compared with the analytical solution.

If the tolerance setting is reduced, it should be expected that the numerical value will more accurately replicate the analytical solution and prove that the convergence mechanism implemented is effective in its task of comparing the numerical iterations.

5.4.5 Interpretation of Numerical Results

Figure 5.4.5 provides a very visual representation of the analytical data (left) presented previously in Figure 5.5.3 with the numerical (right) data simulated on the cluster. It is clear that both shapes hold a strong resemblance to one another however no obvious deviation or anomalies are visible.

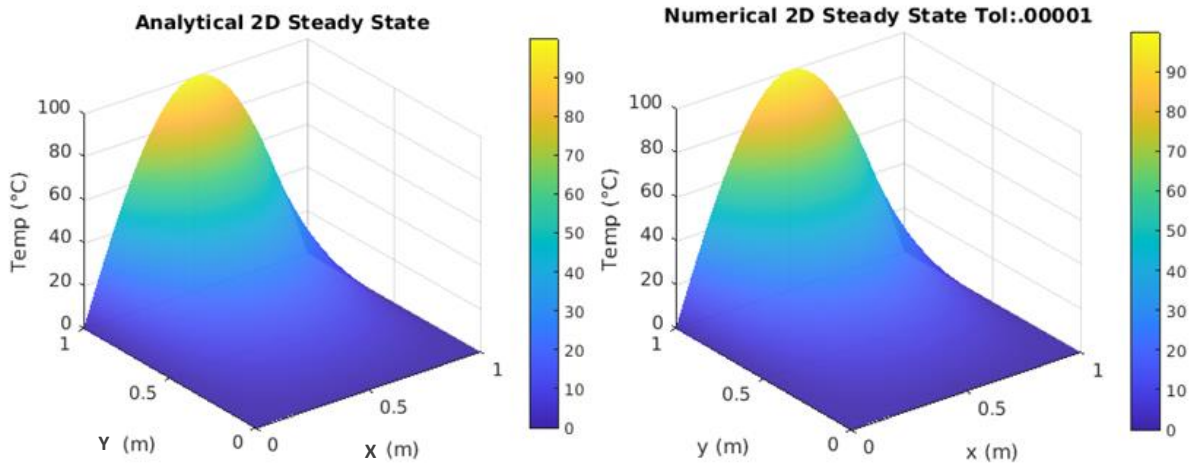

Figure 5.4.5: Visual comparison between surface plot solutions of square grid fixed boundary SS

It was necessary to examine the effects of various tolerance inputs which compare the deviation between two iterative steps of the numerical solution. Figure 5.4.6 shows the contour plot of the same numerical problem solved to varying specified tolerance values. The lower the requested tolerance value, the more accurately these contour plots portray the analytical solution in contour Figure 5.4.3. The temperature is shown propagating though the grid as the number of iterations increases.

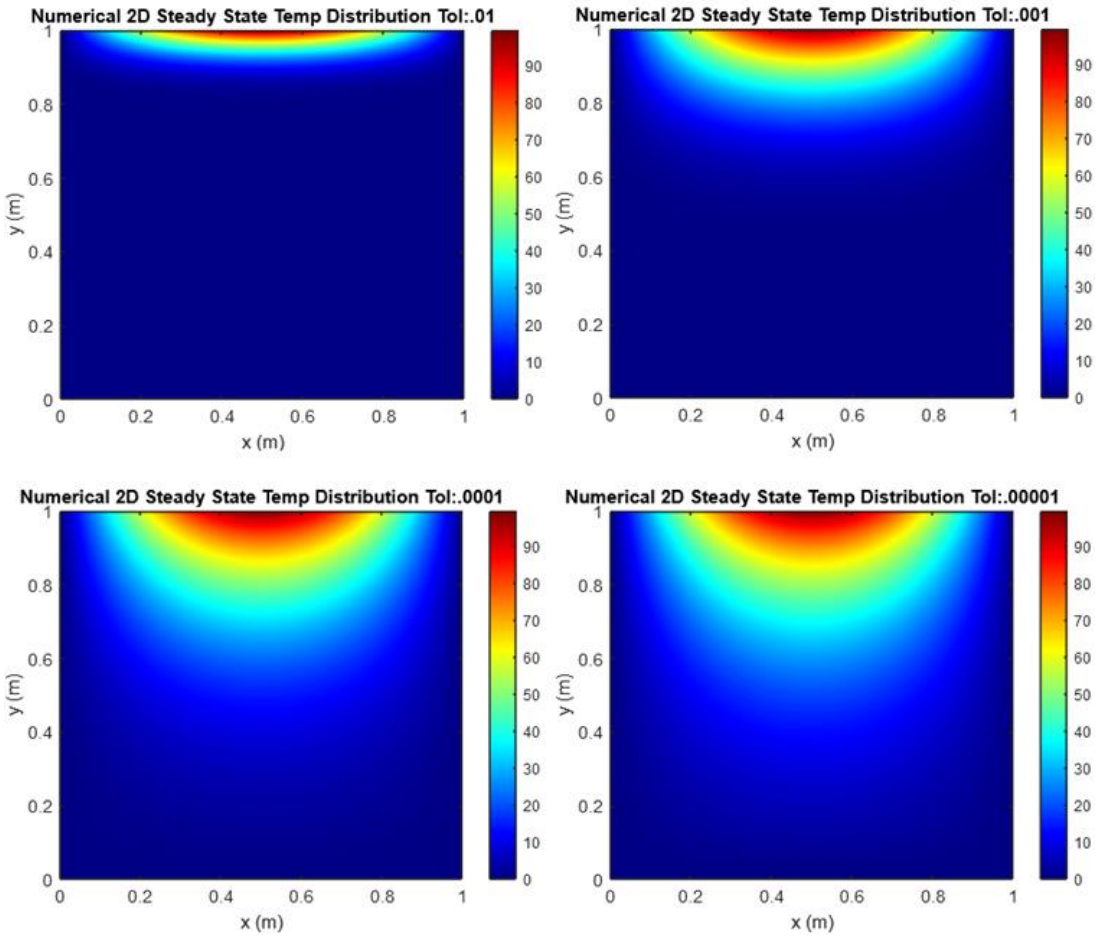

Figure 5.4.6: Numerical solutions of square grid fixed boundary SS with user specified tolerance

The absolute error between the analytical and numerical result is shown in Figure 5.5.7. The error is initially most prevalent near the centre of the top boundary and remains low along all other boundaries. This is because the grid is initialised to a value identical to that of the three; left, right and bottom boundaries, with a shared $T_1 = 0℃$ condition.

The greatest temperature value analytically is expected to appear at the centre of the top boundary causing this error hot spot. This error both reduces and migrates to the centre of the solution for lower tighter convergence criterion; this is a result of the five point stencil shown in Figure 5.4.4. Known information at the boundaries migrates using the stencil one grid node solve at a time, this imposes a sort of information transfer rate. It takes a greater number of iterations for the known boundary information to propagate through the grid and dilute the error of the initial conditions at the centre than near the boundaries.

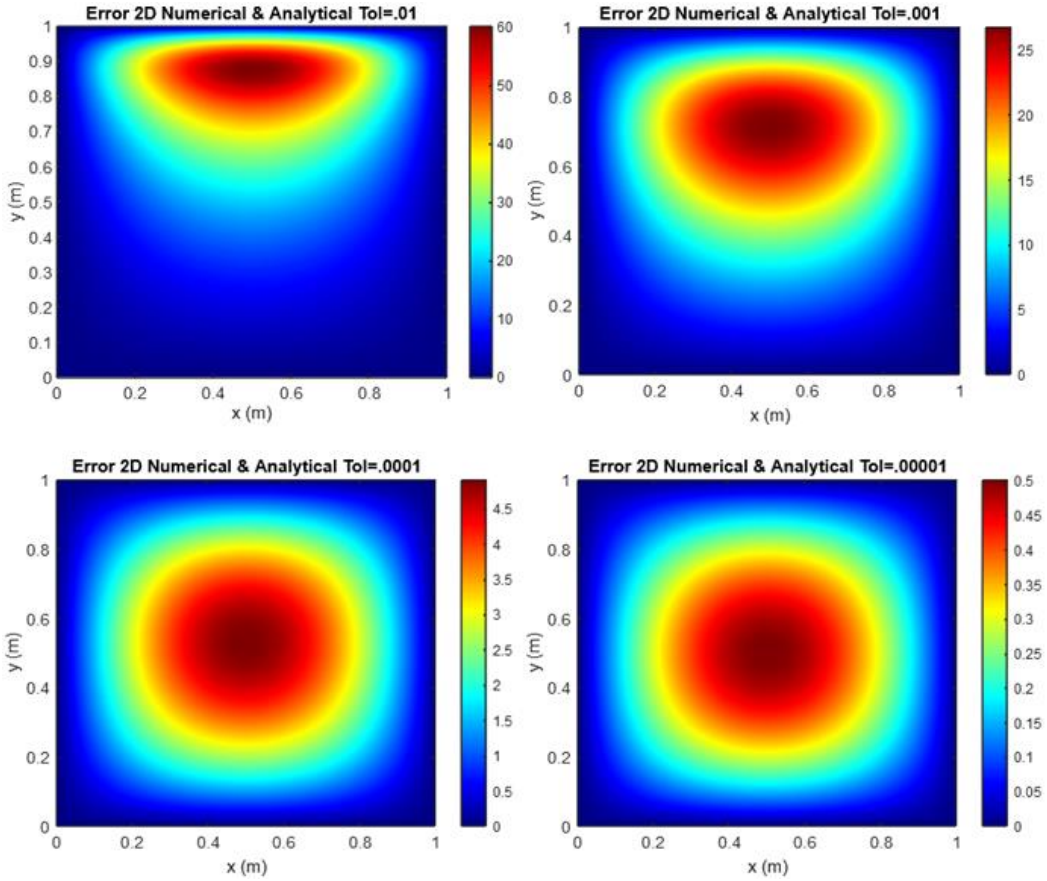

Figure 5.4.7: Error between solutions of square grid fixed boundary SS with user specified tolerance

Figure 5.5.7 shows the error reducing with lower specified tolerance, suggesting the numerical solution is capable of approximating the analytical solution to a desired error given sufficient time. The convergence mechanism used in the numerical calculation was a valid means informing the extent the two solutions agree Table 5.4-1 shows the related run times and number of iterations necessary until the specified convergence tolerance was achieved, there is a non-linear relationship between convergence and the error of the numerical solution.

Table 5.4-1: Convergence tolerance cost

Convergence tolerance	Execution time	Iteration Convergence	Max abs(error)
.01	1.01	2362	60
.001	6.96	19870	27
.0001	32.14	94244	4.8
.00001	69.56	209122	.5

5.5 Annulus 1D Radial Heat Fixed Boundary Steady State

5.5.1 Simulation Conditions

It was decided to simulate an annular structure with dimensions selected to model that of the popular 18650 cylindrical battery cell, which gets its name from its dimensions. The cell has a $diameter = 18mm$ and $height = 65mm$ using Figure 5.5.1: Cross section comparison between the 18650 and 21700 Li-ion cell it was estimated that the internal annular radius $r_i = .002m$ and outer $r_o = .009m$.

The annulus is periodic as explained previously with Figure 4.3.2; there are only two boundaries for the problem, located at the inner and outer radial surfaces. The problem will examine a 2D grid however will only apply a 1D heat conduction in the radial dimension, It will be assumed that this is an **anisotropic** material where conduction is only possible radially as such the temperature is only dependant on the radial position $T(r)$ however for the sake of a more interesting visual graph the internal boundary condition will be variable in the tangential direction resulting in an apparent tangential dependency on the values of the internal temperatures T_i. What this equates to is performing many 1D heat conduction simulations along 1D bars with different boundary conditions. Then simply presenting them offset from one another by an angle as a 2D grid.

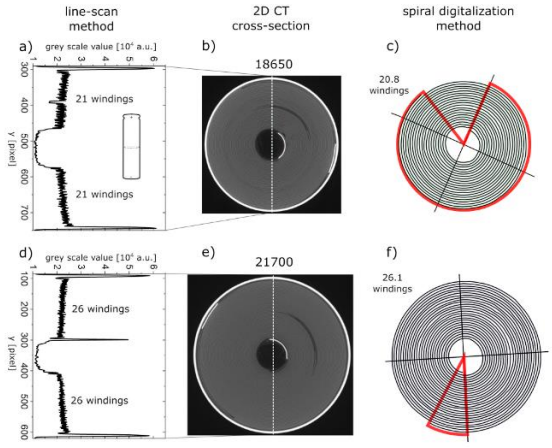

Figure 5.5.1: Cross section comparison between the 18650 and 21700 Li-ion cell [11]

The fixed boundary conditions were provided shown in Figure 5.5.2 where a constant temperature exists on the outer boundary $T_o = T(0) = T_1 = 20℃$ and a variable internal surface temperature is $T(a, \emptyset) = T_o + T_m \sin\left(\frac{\pi\emptyset}{2\pi}\right)$, with $T_m = 100℃$. Figure 5.5.2 provides a visual representation of the cross section and its relevant nomenclature, where the radial step size $\delta = \frac{b-a}{N}$ [3].

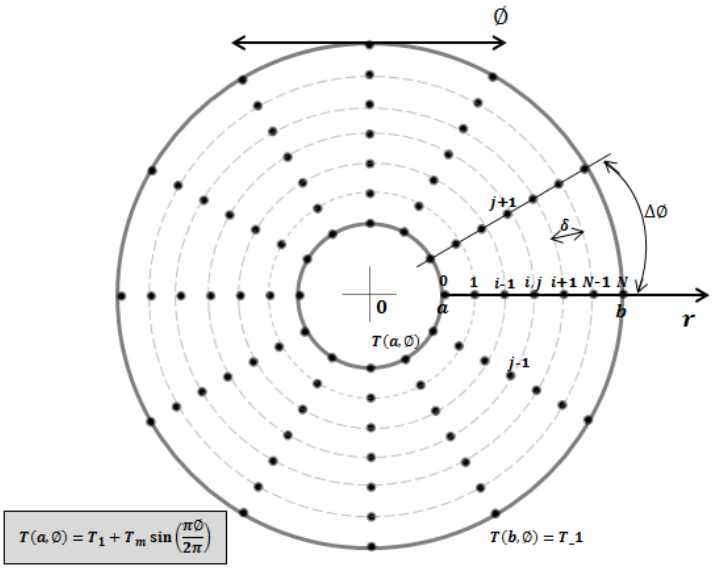

Figure 5.5.2: Annular 1D radial heat fixed boundary SS problem nomenclature

5.5.2 Analytical Solution

One can start with the general solution for the Laplacian shown in (5.5-1), applied to a cylindrical shaped object where r represents the radial axis, Theta ($\emptyset$) the tangential axis and z the axial axis. The tangential axis is periodic with a period of 2π.

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial T}{\partial r}\right)+\frac{1}{r^2}\frac{\partial^2 T}{\partial \emptyset^2}+\frac{\partial^2 T}{\partial z^2}+\frac{1}{k}g(r,\emptyset,z)=\frac{1}{\alpha}\frac{\partial T}{\partial t} \quad (5.5\text{-}1)$$

In this case there is no internal heat generation, the system runs to determine the steady state solution with one dimensional heat transfer in the radial direction so the function can then be reduced to (5.5-2).

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial T}{\partial r}\right)=0 \quad (5.5\text{-}2)$$

Integrating the second order equation (5.5-2) gives the expression equation (5.5-3).

$$T=C_1\ln(\mathrm{r})+C_2 \quad (5.5\text{-}3)$$

Two constants persist, however for this problem there are known boundary condition temperatures internal T_i and outer T_o located at r_i and r_o respectively. These can be subbed into the expression obtained to re-arrange and constants in terms of the known boundary values.

$$\begin{matrix} T_i=C_1\ln(r_i)+C_2 & => \\ T_o=C_1\ln(r_o)+C_2 & => \end{matrix} \begin{cases} C_1=\dfrac{T_i-T_o}{\ln\left(\frac{r_i}{r_0}\right)}=\dfrac{\Delta T}{\ln\left(\frac{r_o}{r_i}\right)} \\ C_2=T_i+\dfrac{\Delta T}{\ln\left(\frac{r_o}{r_i}\right)}\ln(r_i) \end{cases} \quad (5.5\text{-}4)$$

The expressions for the constants can now be subbed back into (5.5-4(5.5-3).

$$T=\left(-\frac{\Delta T}{\ln\left(\frac{r_o}{r_i}\right)}\right)\ln(\mathrm{r})+\left(T_i+\frac{\Delta T}{\ln\left(\frac{r_o}{r_i}\right)}\ln(r_i)\right) \quad (5.5\text{-}5)$$

$$T=T_i-\frac{\Delta T}{\ln\left(\frac{r_o}{r_i}\right)}\ln(r-r_i) \quad \rightarrow \quad T=T_i-(T_o-T_i)\left(\frac{\ln\left(\frac{r}{r_i}\right)}{\ln\left(\frac{r_o}{r_i}\right)}\right) \quad (5.5\text{-}6)$$

Which means that for any desired position r where $r_i<r<r_o$ the corresponding value of temperature T can be obtained simply by substituting the known values and grid positions. [12]

Mathlab code was written to apply this analytical solution to generate a 500x500 meshed grid and plot visual representations of the heat distribution over the 2D surface. By substituting the same conditions used into a numerical solution the difference of the two can be checked to verify the suitability of the numerical solution obtained.

5.5.3 Interpretation of Analytical Results

Figure 5.5.3 shows a 3D surface plot of the temperature distribution obtained from running the analytical calculations. The internal boundary is fixed and was assigned with tangential dependency. The temperature is periodic increasing from a minimum $T_1=20$ to a maximum T_m+T_1=120 with a half a period π. The periodic nature as explained previously with Figure 4.3.2 means there are only

two boundary conditions, the tangential surface would otherwise show two radial boundaries however the surface is smooth and uninterrupted. The 1D heat conduction in the radial dimension is seen to dissipate radially towards the outer boundary temperature. Where the internal boundary has the highest temperature the surface slope is greatest, this is in stark comparison with the lowest internal temperature which matches the external temperature resulting in a flat surface as no potential difference exists between the inside and outside surface to cause heat conduction to occur.

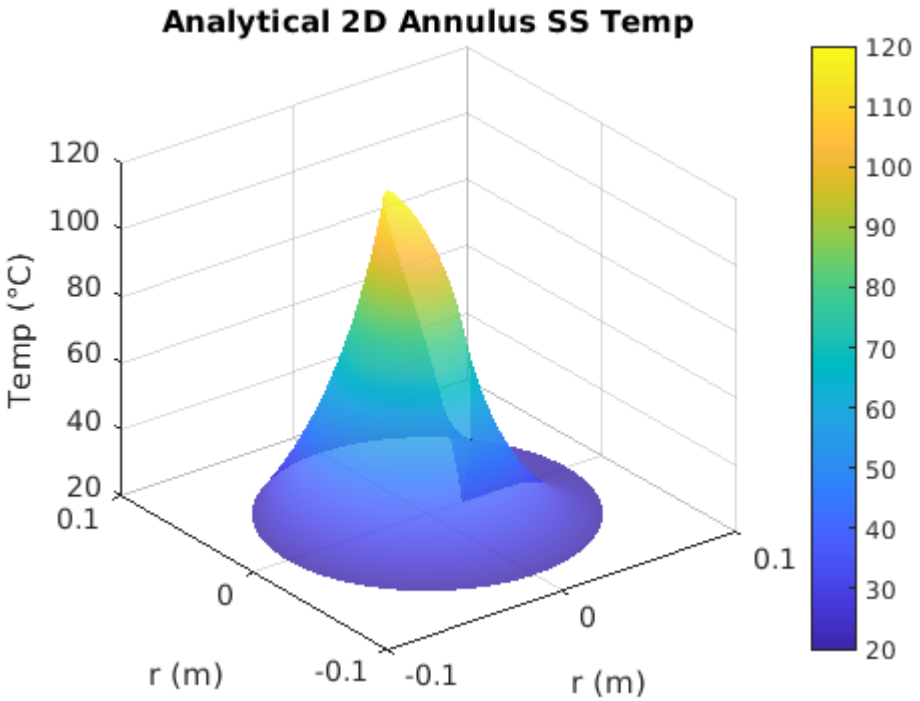

Figure 5.5.3: Surface plot of analytical solution of annular grid fixed boundary SS

It is somewhat obscured in the surface plot but the annular shape can be shown more clearly in a contour plot Figure 5.5.4 and matches the desired radial dimensions specified.

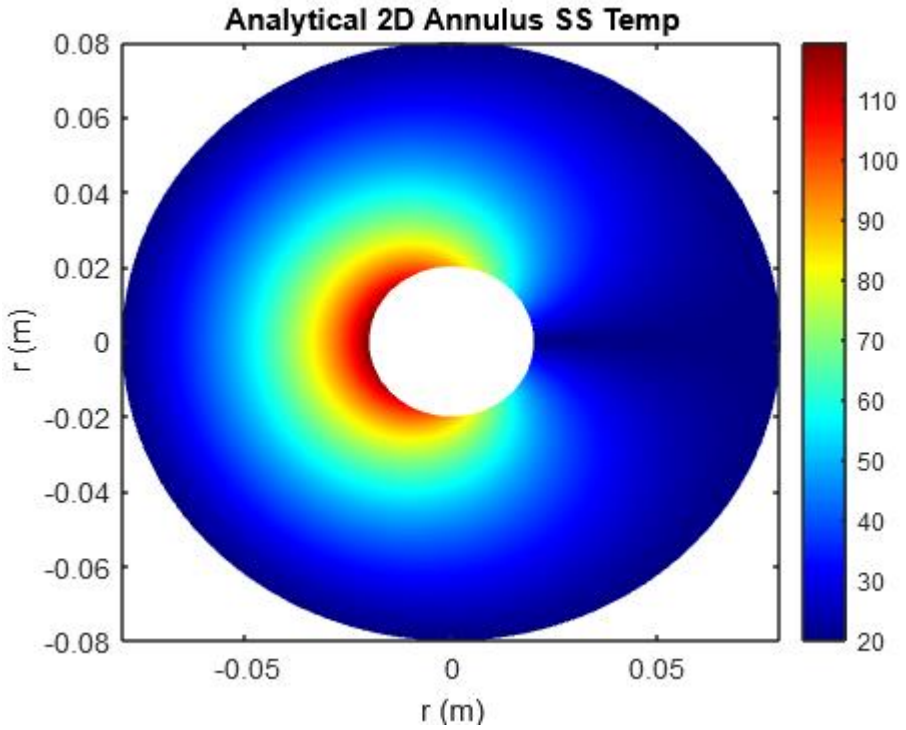

Figure 5.5.4: Contour plot of analytical solution of annular grid fixed boundary SS

5.5.4 Numerical Solution

An expression for calculation of the iterative numerical solution can be obtained from the general solution of the Laplacian shown in (5.5-7) accounts for the polar co-ordinate system within a cylindrical shaped object.

$$\frac{d^2T}{dr^2}+\frac{1}{r}\frac{dT}{dr}+\frac{1}{r^2}\frac{d^2}{d\emptyset^2}+\frac{1}{k}g(r)=0 \qquad (5.5\text{-}7)$$

In this case the steady state solution for one dimensional heat conduction is desired, and the time and z dimensions can be dropped to leave the formula (5.5-8). The central difference method can then be applied to the both first and second order derivatives as shown (5.5-9).

$$\frac{d^2T}{dr^2}+\frac{1}{r}\frac{dT}{dr}=0 \qquad (5.5\text{-}8)$$

$$\frac{T_{i-1,j}-2T_{i,j}+T_{i+1,j}}{\Delta r^2}+\frac{1}{i\Delta r}\frac{T_{i+1,j}-T_{i-1,j}}{2\Delta r}+\frac{1}{k}g(r)=0 \qquad (5.5\text{-}9)$$

$$where:\ a\le r<b, \qquad 0\le\emptyset\le 2\pi$$

The radial step size can be expressed in terms of the known (user specified) boundary dimensions. It is important to note when examining the code in file 9.1-21 a translation must occur from the provided visual Figure 5.5.2 and equations. Due to the existence of ghost column/rows the array position holding the value of the inner radius will begin at position one as opposed to zero shown, this means the step size in the radial direction will be over N-1 instead.

$$\Delta r = \delta = \frac{b-a}{N} \qquad r_i = a \neq 0, \qquad r_o = b > r_i \tag{5.5-10}$$

$$\frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{\delta^2} + \frac{1}{a+i\delta}\frac{T_{i+1,j} - T_{i-1,j}}{2\delta} + \frac{1}{k}g(r) = 0 \tag{5.5-11}$$

The equation can then be manipulated to solve for a future value of $f_{i,j}$ using previous iteration values as input (Jacobi Iteration) and in the case of the very first calculation using assumed input values.

$$T_{i-1,j} - 2T_{i,j} + T_{i+1,j} + \frac{\delta^2}{a+i\delta}\frac{T_{i+1,j} - T_{i-1,j}}{2\delta} + \frac{\delta^2}{k}g(r) = 0 \tag{5.5-12}$$

$$T_{i-1,j} - 2T_{i,j} + T_{i+1,j} + \frac{\delta^2}{a+i\delta}\frac{T_{i+1,j}}{2\delta} - \frac{\delta^2}{a+i\delta}\frac{T_{i-1,j}}{2\delta} + \frac{\delta^2}{k}g(r) = 0 \tag{5.5-13}$$

$$T_{i-1,j} - 2T_{i,j} + T_{i+1,j} + \frac{\delta^2 T_{i+1,j}}{2(\delta a + \delta^2 i)} - \frac{\delta^2 T_{i-1,j}}{2(\delta a + \delta^2 i)} + \frac{\delta^2}{k}g(r) = 0 \tag{5.5-14}$$

$$T_{i-1,j} - 2T_{i,j} + T_{i+1,j} + \frac{T_{i+1,j}}{2\left(\frac{a}{\delta}+i\right)} - \frac{T_{i-1,j}}{2\left(\frac{a}{\delta}+i\right)} + \frac{1}{k}g(r) = 0 \tag{5.5-15}$$

$$-2T_{i,j} + \left(1 + \frac{1}{2\left(\frac{a}{\delta}+i\right)}\right)T_{i+1,j} + \left(1 - \frac{1}{2\left(\frac{a}{\delta}+i\right)}\right)T_{i-1,j} + \frac{1}{k}g(r) = 0 \tag{5.5-16}$$

$$T_{i,j} = .5\left(\left(1 + \frac{1}{2\left(\frac{a}{\delta}+i\right)}\right)T_{i+1,j} + \left(1 - \frac{1}{2\left(\frac{a}{\delta}+i\right)}\right)T_{i-1,j} + \frac{1}{k}g(r)\right) \tag{5.5-17}$$

Equation (5.5-17) was then used to solve for internal (non-boundary) grid points in the *Sweep_Solve_1D_SteadyState_Annulus_FixedBound( )* function available within file 9.1-2.

The problems boundary conditions were written into a designated file 9.1-21 with the specified grid dimensions. This was then compiled using –xSSE4.2 and -03 optimization flags and executed for a 500x500 grid with a maximum iteration of 500,000 (high enough to ensure convergence occurred). The problem was run on the Kelvin cluster across 4 processors, although it will be shown in a scalability testing section that the number of processors has no impact on the numerical values obtained. The program was run once with a specified $tolerance = .0001$, as a convergence requirement for comparison against the analytical solution.

5.5.5 Interpretation of Numerical Results

Figure 5.5.5 provides a visual representation of the numerical data simulated on the cluster. It is clear that the plot produced shows a strong similarity to the analytical solution Figure 5.5.3 with no obvious deviation or anomalies visible. The maximum and minimum temperatures at the boundaries are correct and the points spatially between pose temperatures between these two values.

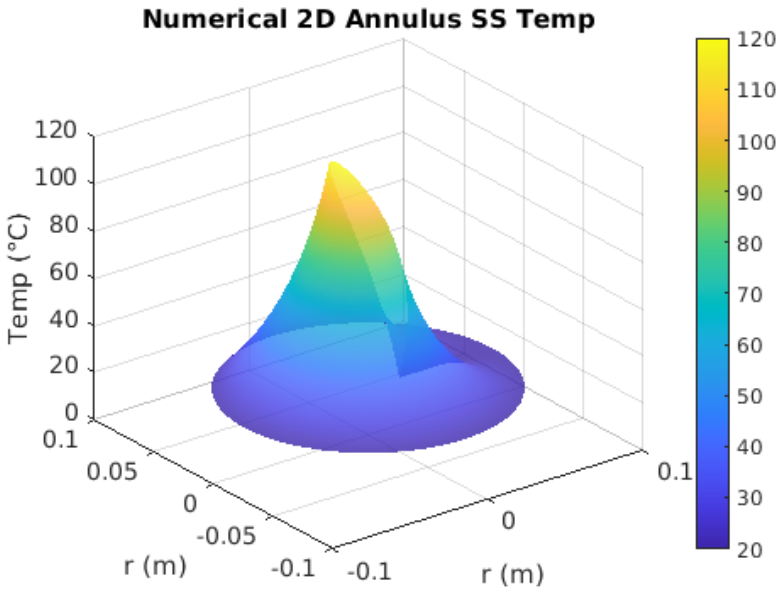

Figure 5.5.5: Surface plot of numerical solution of annular grid fixed boundary SS

Similarly the dimensions on the radial axis appear correct in Figure 5.5.6 with a very similar thermal distribution to that seen in the analytical solution with temperature dissipating radially from the high internal boundaries to the low external value.

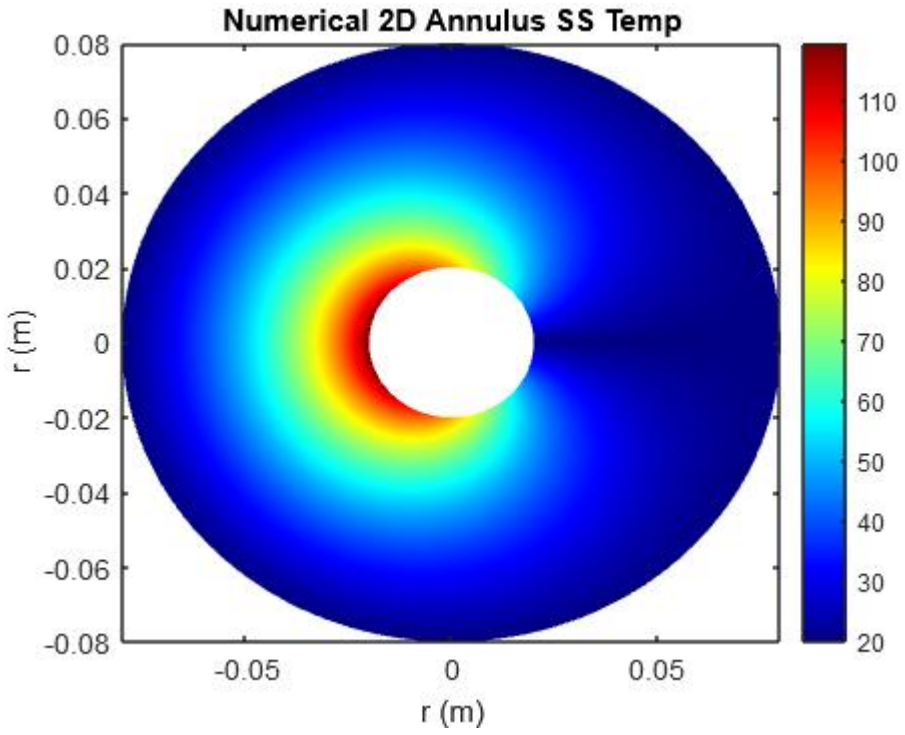

Figure 5.5.6: Contour plot of numerical solution of annular grid fixed boundary SS

A plot comparing the absolute error between the analytical and numeric solutions is shown in Figure 5.5.7. The error is greatest on the side of greatest temperature gradient between the inner and outer boundary while no error exists between the right hand side, where a minimum internal temperature matches the external boundary. The reason for this error distribution is again because the grid was initialised to the T_1 value and so areas with greater boundary temperatures were more prone to initial

deviate from this. The area of greatest error appears to hug the internal diameter on the left side of the plot, this error would be expected to approach the radial midpoint value between $r_i$and r_o for the same reasons discussed in the square grid example. The magnitude of the error and location suggest that the simulation would need to run over a greater number of iterations to achieve an error of similar characteristics to that shown in the square example for the same tolerance value.

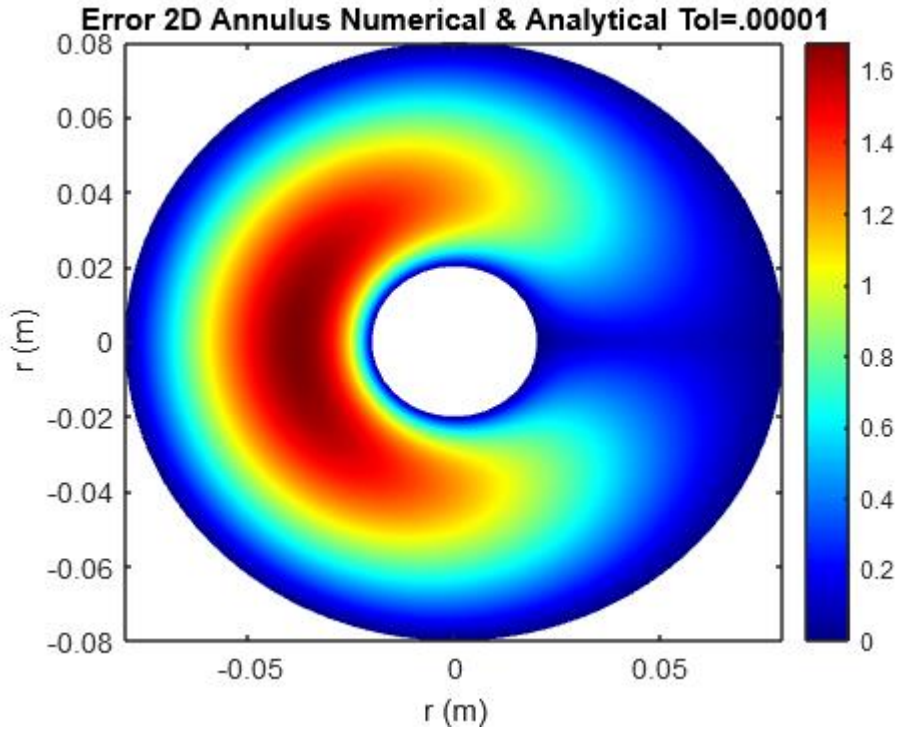

Figure 5.5.7: Error plot of analytical vs numerical solution of annular grid fixed boundary SS

5.6 Strong Scaling

5.6.1 Test Procedure

A strong scaling performance test was carried out by examining the effects of distributing a fixed problem size across an increasing numbers of processors. A square of dimensions $x = 1m,\ y = 1m$ was discretised into a grid of $M = 1000,\ N = 1000$ nodal postitions. Fixed boundary conditions were applied to solve a 2D Cartesian steady state problem with a specified convergence criterion $tolerance = .001$, using file 9.1-1 (Simulated problem similar to Section 5.4).

Execution time was obtained using the MPI Library *MPI_Wtime( )*; functions located at the start and end of the simulation file 9.1-1. The input parameters were kept identical for all runs (fixed problem) with the only variable the number of processors, which were incremented in powers of 2 from 1-6 to distribute over 1-36 processors. By incrementing in these powers of 2 the grids can be guaranteed to distribute evenly the quantity of gird rows/columns on each processor. This was performed on the Kelvin cluster using an sbatch script similar to file 9.1-11 with the problem was compiled prior using level 3 optimisation with a -03 flag along with –xSSE4.2 to specify the use of hardware specific vector extensions

The concept of strong scaling comes from Amdahl's law which states that the program run-time can be broken into two components of both the serial (s) and parallel portion (p). When run over multiple processors only the parallel portion stands to provide any improvement in the performance. As the number of processors approaches infinity the serial portion limits the rate of speed up.

$$t_1 = s + p \tag{5.6-1}$$

$$t_n = s + \frac{p}{N} \tag{5.6-2}$$

$$P_1 = \frac{s + p}{t_1} \tag{5.6-3}$$

$$Speedup = \frac{t_1}{t_n} = \frac{N}{s + \frac{1 - s}{N}} \tag{5.6-4}$$

[4]

A measure of the serial proportion can be obtained with experimental values for execution over a sufficiently large number of processors. This expression can be obtained through manipulation of Amdahl's law to the expression shown in (5.6-6).

$$\lim_{n \to \infty} \frac{t_1}{t_n} = \frac{1}{1 - p}, \quad \text{Amdahl's Law } as\ N \to \infty \tag{5.6-5}$$

$$\lim_{n \to \infty} \frac{t_1}{t_n} = \frac{1}{s} \tag{5.6-6}$$

Of course the purpose of parallelising the code and distributing over multiple processors is to obtain the same solution in a faster time. With strong scaling it is important that the problems solved are identical and the same total work was done. In order to verify that the resulting grids solved in parallel matched the serial solution, Mathlab code was written to import the output decimal.txt from the first serial run (1 processor) and the final parallel execution (over 36 processors) before comparing error between them on a grid node by node basis.

5.6.2 Interpretation of Results

Table 5.6-1 provides the resulting execution times obtained from the experiment, using these times for the single and the multi-processor solutions the serial and parallel fractions were calculated using (5.6-6). Figure 5.6.1 shows the execution times taken with respect to the number of processors the problem was run on. A trend line was fit to the data and it is clear from this that as the quantity of processors is increased the effect on execution time diminishes by a power of -.778.

Table 5.6-1: Experimental strong scaling results

Procs:	**1**	**4**	**9**	**16**	**25**	**36**
Execution time (s):	159.19	45.53	27.77	15.25	12.08	10.11
Speedup:	1	3.50	5.7	10.44	13.18	15.75
Grid Size (nodes):	1000000	1000000	1000000	1000000	1000000	1000000
Serial portion:	1	.285	.175	.096	.076	.063
Parallel portion:	0	.715	.825	.904	.924	.937

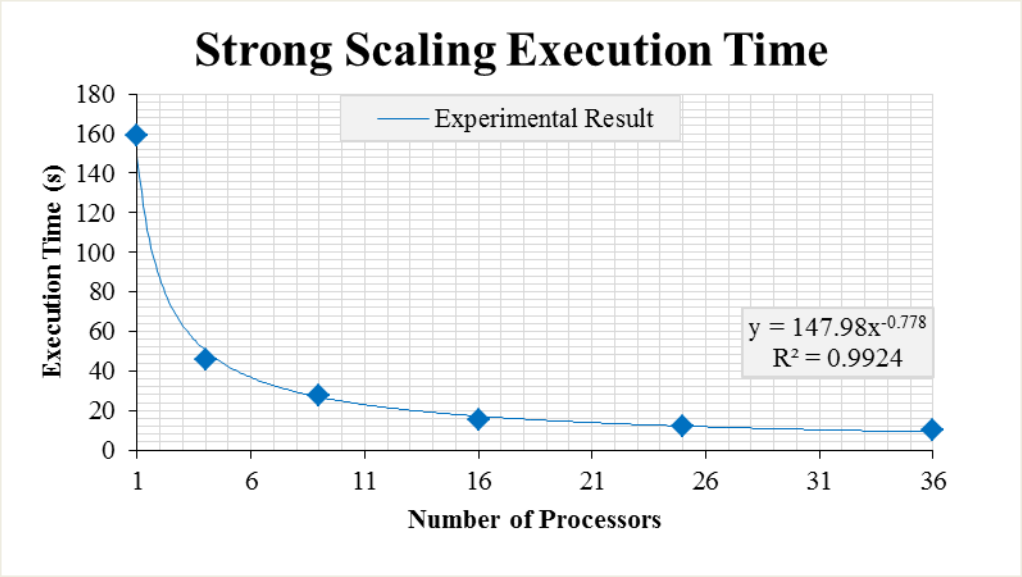

Figure 5.6.1: Strong scaling of the 2D_Rect_Fixed_Boundary_SS problem

The results obtained in Figure 5.6.2 show the expected bottlenecking incurred by the non-scalable serial portion of the code which causes the experimental results to deviate from the theoretical 1:1 perfect scaling. The speedup rapidly diminishes with the increase in processors. In both graphs trend lines were fitted which achieved high R^2 values, indicating a low variance in the experimental data and no abnormal outliers. From this, one could argue that running the program on more than 9 to 16 processors would be a wasteful use of resources as the gained speedup becomes negligible.

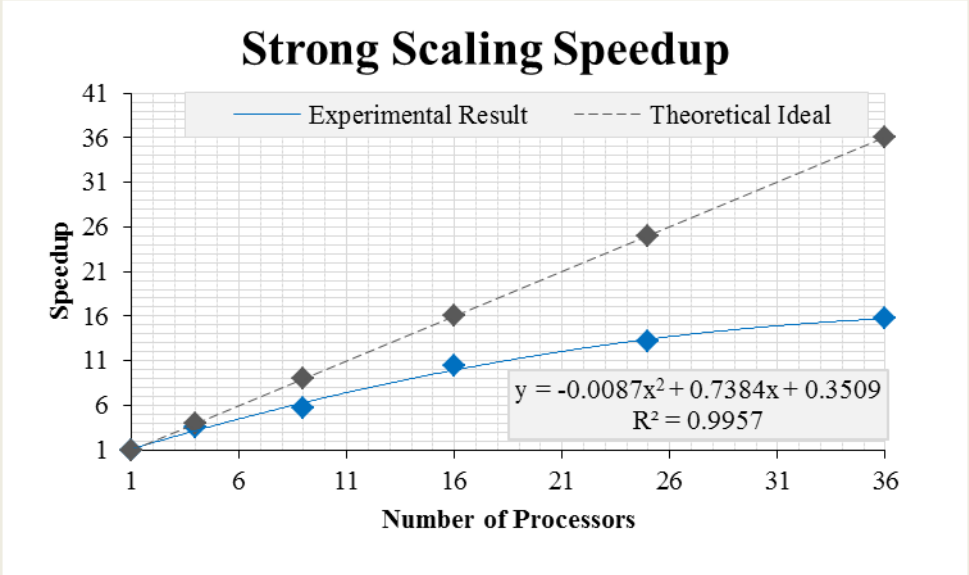

Figure 5.6.2: Strong scaling speedup of the 2D_Rect_Fixed_Boundary_SS problem

It was important to ensure that the results obtained for the serial and parallel implementations were identical for the sake of the strong scaling results. The purpose of parallelizing code is to obtain the same result in a shorter time period. Figure 5.6.3 although seemingly not very interesting is significant as it shows absolutely no error between the output result when run in serial vs in parallel. This result does not indicate whether the numerical method itself has obtained an accurate solution however serves as validation of the codes ability to distribute the numerical problem successfully across processors and gather the solution again at the end thus successfully parallelising the problem.

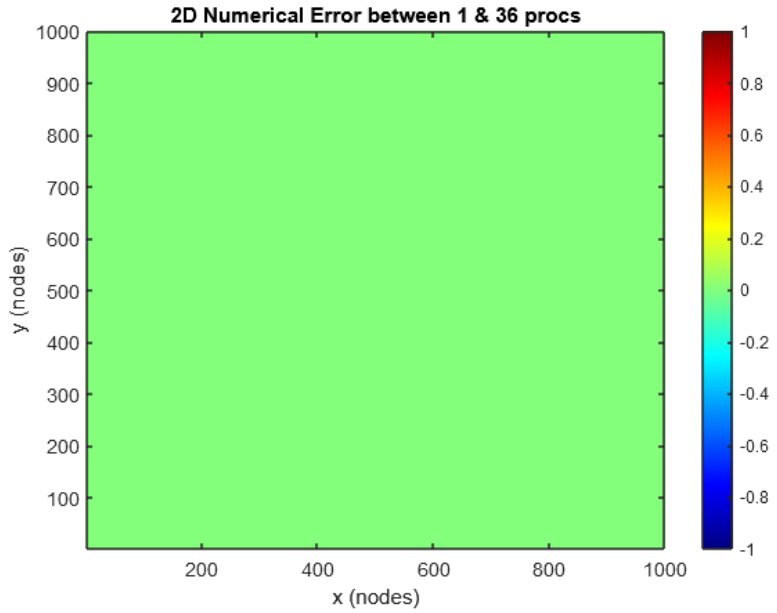

Figure 5.6.3: Error between single vs multi proc numerical solution

5.7 Weak Scaling

5.7.1 Test Procedure

Weak scaling was performed for a similar 2D Cartesian steady state problem to the strong scaling example by maintaining the same workload for each processor as the number of processors were increased. Similarly, input parameters were kept consistent throughout with the only variability; the grids resolution and number of processors, incremented in powers of 2 over 1-6 for 1-36 processors. The same square grid of dimensions $x = 1m,\ y = 1m$ was used however the number of grid nodes were increased each time with the addition of processors to ensure a $500x500$ grid would be allocated per processor.

In this case the weak scaling speedup, calculated using (5.7-1), considers of the serial time it would take to solve a grid size equivalent to that solved on multiple processors.

$$Speedup = \frac{N \times t_1}{t_n(N)} \tag{5.7-1}$$

The convergence checking mechanism was removed for this test and instead the only stopping criterion for each execution was to perform 5,000 iterations of the sweep. This was to ensure each execution maintained an equivalent amount of work per processor; otherwise some executions would converge and result in fewer iterations of the sweep.

This was executed on the Kelvin cluster using a sbatch script and the problem, file 9.1-1 compiled prior using level 3 optimisation with a -03 flag. The use of vector extensions for improved vector optimizations was again applied using –xSSE4.2 flag.

Using Gustafson's law an expression (5.7-2) was obtained for determining the serial fraction an consequently the parallel fractions of the code.

$$Speedup = s + p \times N \tag{5.7-2}$$

$$Speedup = s + N - sN \tag{5.7-3}$$

$$\frac{Speedup - N}{1 - N} = s \tag{5.7-4}$$

[13]

In an attempt to isolate the speedup result from the effects of the *Parallel_write_to_file( )* function, the procedure was repeated with recompiled code where the function call was commented out. Two situations were examined, where the grid size stayed the same per processor (500 x 500) and another where it was increased by a factor of (5,000 x 5,000) per processor.

5.7.2 Interpretation of Results

Weak scaling was initially performed using problem grids of 250,000 nodes allocated per processor, the resulting speedup is shown visually in Figure 5.7.1 as the orange data points. The speedup achieves the expected linear relationship with processors however the slope compared to the theoretical ideal seemed quite poor. The fitted linear trend line for the same has a less than ideal R^2 value indicating a possible inconsistency in the result and the line doesn't appear to intersect the x axis.

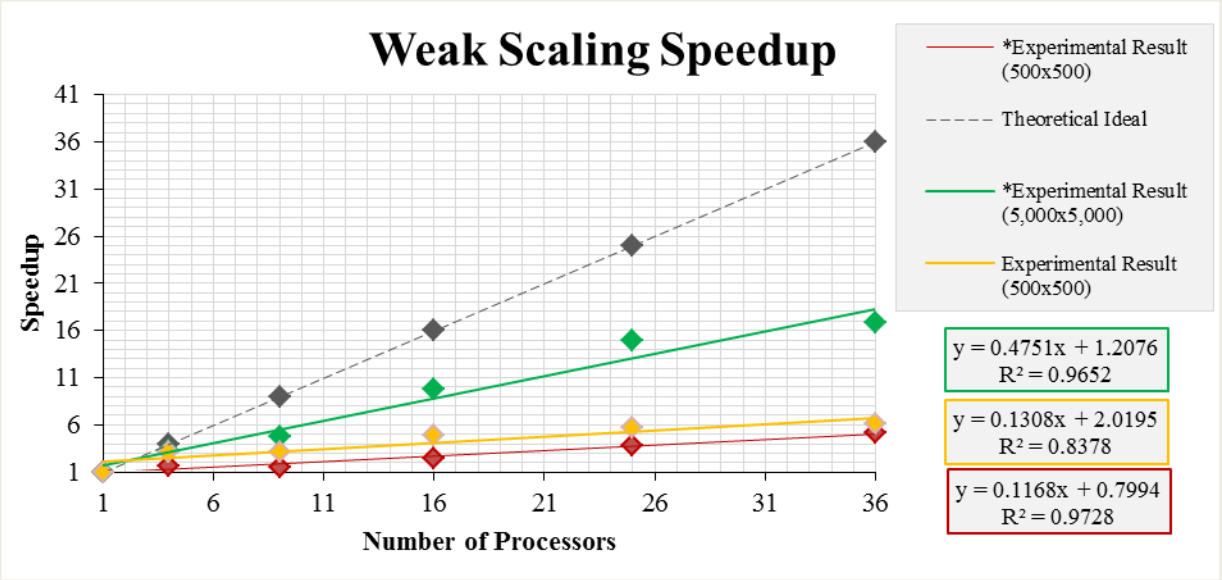

Figure 5.7.1: Weak scaling speedup of the 2D_Rect_Fixed_Boundary_SS problem

This encouraged further investigation into the cause of the poor performance, in an attempt to troubleshoot the cause, the problems were re-run with the *Parallel_Write_to_file( )* function removed. The results of this test are shown on the same Figure 5.7.1 in red and formed a very similar slope however intersection with the x axis occurred and an improved R^2 value was achieved. Because these results are normalised to T_1 it is useful to review results Table 5.7-1 to see the actual execution times for the performed tests. While a slight improvement in runtime might have been expected at the

removal of the function, this reveals a highly significant improvement in run time across all processors.

Table 5.7-1: Experimental weak scaling results

No. Processors:	1	4	9	16	25	36
Execution time (s):	64.3	80.73	182.59	208.46	284.72	380.71
Speedup:	1	3.19	3.17	4.94	5.65	6.08
Grid Size (nodes):	250000	1000000	2250000	4000000	6250000	9000000
***Execution time (s):**	6.97	17.28	43.79	44.53	46.23	49.13
***Speedup:**	1	1.61	1.43	2.50	3.77	5.11
Grid Size (nodes):	250000	1000000	2250000	4000000	6250000	9000000
Time Change (s):	57.33	63.45	138.8	163.93	238.49	331.58
***Execution time (s):**	959.9	1181.88	1803.37	1580.79	1610.94	2054
***Speedup:**	1	3.25	4.79	9.72	14.90	16.82
Grid Size (nodes):	25,000,000	100,000,000	225,000,000	400,000,000	625000,000	900,000,000

* **W**ithout *Parallel_print_to_file()* function

To delve further the differences between these two runs were calculated and are displayed in Figure 5.7.2. This shows that not only does the function incur a large runtime cost across all processors but the cost does not remain uniform with the increase in the quantities of processors but rather, also increases. A linear relationship is shown increasing with a slope of almost 8s/proc.

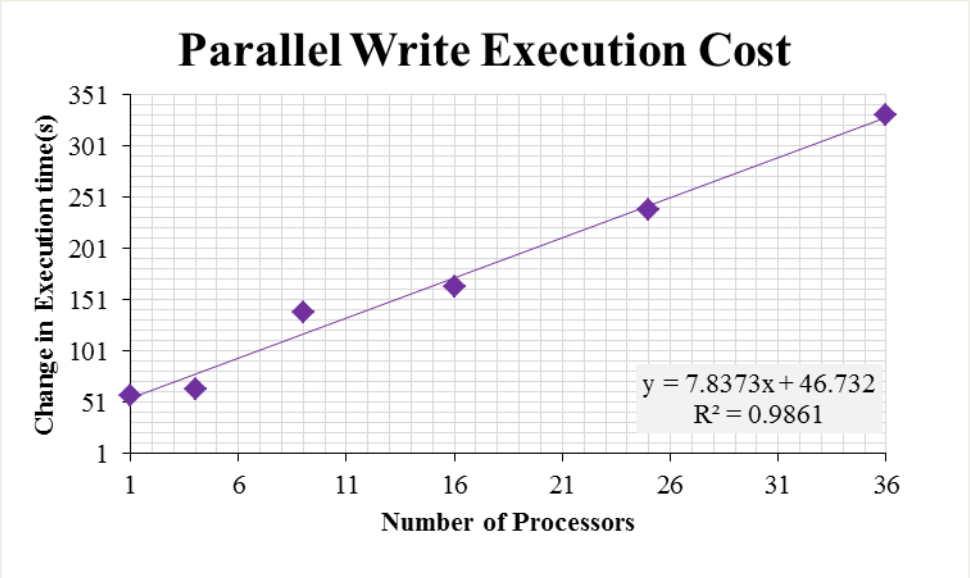

Figure 5.7.2: Additional cost of Parallel_write_to_file() function on weak scaling

This came as a surprise as the purpose of the *Parallel_write_to_file( )* function was to enable each processor independent access to the same file and to write simultaneously, in theory resulting in a similar execution time regardless of the number of processors (when applied to weak scaling). Because of this result, the MPI library functions within the *Parallel_write_to_file( )* function were investigated further. This revealed that while the *MPI_write( )* function does allow simultaneous printing to file the other *MPI_File_open( )* and *MPI_File_Close( )* are actually a type of collective communication. This means that before any one processor is allowed to open or close the file, all other processors must also have reached the same command.

These highlighted two inefficiencies, all processors must wait on the slowest processor before any can begin printing to file. The second is that of the file pointer offset which has an unequal volume of

work to do depending on the position in the output file the processor is due to print its local grid block. This is exacerbated with weak scaling as the work loads are supposed to be equal however the printing to file creates an inherent inequality as the global grid size is increasing with the increase in processors. Because they all begin writing at the same time processor on rank 0 can begin printing immediately in the first file pointer position however the last processor is required to perform an additional offset (step) through the file positions until it reaches its offset location near the end of the file.

Finally all processors must wait on the slowest processor before closing the file. These collective calls are used in MPI for error checking purposes to ensuring the file successfully opens/for all processors. As none of the processors have any need for inter dependency opening, writing and closing the file it may be possible to implement a non-collective type of MPI call instead. This would allow some minor improvements in execution as the slowest processor to open a file has the opportunity to redeem lost time in the printing and closing stages.

If one considers again the orange trend line on Figure 5.7.1, it likely doesn't cross the x-axis because the serial run is not subjected to the same wait requirements as other runs before printing. It sees no slowdown from pointer offsets which causes a skew the linear relationship.

A run is shown in green on Figure 5.7.1 which used an increased 25,000,000 grid nodes per processor. This problem size resulted in a significant improvement in the weak scaling slope, providing much better speedup performance when greater numbers of processors were used than previous. The reason for this improvement is suspected to be caused by the trade-off between the *Exchange_Data_2D ( )* necessary for parallel communication and the *Sweep_Solve_2D( )* which performs the iterative data. The grids need to be sufficiently large to obtain good weak scaling. The parallel communication expense cripples the speedup result if there is not sufficient work to be done on the data by each processor. This is the purpose of performing weak scaling, it indicates the point at which a problem is worthy of additional computation resources. If the number of grid nodes were increased again the weak scaling would be expected to better approach the theoretical ideal shown in grey.

An anomaly was noticed in the resulting execution time shown in Table 5.7-1 for nine processors. This result appears to possess a large jump in the run time from four to nine processors, particularly if one considers the similar execution times obtained from nine to sixteen processors. One possible influence on this anomaly may be the hardware explained previously, Figure 3.7.1 indicates that each compute node on the cluster is made up of two sockets of six processors. The need to communicate between processors on a different socket as the number of processors exceeds those available on single socket could incur an additional runtime expense; this may accumulate over the number of iterations causing this time expense jump.

A second possible influence is a fact that was only realised following the weak scaling tests. The work done by each processor is not equal despite the fact that each are allocated the same number of grid nodes. This is because processors which possess the boundary conditions have one less row/column to perform a sweep calculation on and one less direction to communicate for every boundary condition. This means that a single processor performs $x4$ less exchange communications and $x4$ less column/data solves to a processor allocated no boundary conditions. Similarly when run on four processors each processor holds two boundaries and so has $x2$ less exchange communications and $x2$ less row/column solves to perform.

The unequal distribution of workload is unavoidable with the method used to distribute and solve the global grid. The code was written in this way to enable both ghost rows/columns and the ability to include the boundary values in the printed global grid using the *Parallel_Print_to_file( )* function.

6. Summary

Although not stated as an objective, a strong initial focus was placed on writing utility functions for distributing the work, allocating local memory and parallel printing. It had been assumed the creation of a binary to decimal conversion tool was a well-documented problem; however developing the small python script specifically for the double precision case was a time consuming task someone unfamiliar with the language. The utility functions were followed by code to create a working Cartesian model with 2D heat conduction. This problem was first performed to demonstrate the code would operate as desired for a simpler problem and to compare with an analytical model before continuing.

It was demonstrated that the finite difference model could be successfully used to approximate the analytical result with an error shown to reduce with increased number of iterations using a specified convergence criteria.

From this point the first objective, simulation of the annular style geometry was approached. To compare this with an analytical solution and verify successful simulation this was simulated using only 1D heat conduction and not the initial 2D objective stated. This simulation was found to successfully approximate the analytical problem. The code written is capable of simulation of this geometry with 2D conduction by simple manipulation of the function written within *Sweep_Solve_1D_SteadyState_Annuluss_FixedBound(*) to accommodate the inclusion of a tangential relationship however no such implementation was demonstrated in this report.

An approach was applied to feature convective boundary types for the steady state, annular, 1D heat conduction simulation. This included additional boundary specific calculation updates within the iterative sweep calculation function. This produced results which rapidly approached infinity and not the ambient temperatures specified. Despite troubleshooting no cause or resolution for this was discovered. Similarly an attempt was made to write a transient simulation however there was insufficient time to complete troubleshooting this code and obtain results to compare against analytical solution.

A final fourth objective, for internal heat generation was never applied. It is understood that this in theory only involves the inclusion of an additional term to the pre-existing formula within the sweep functions. No such simulations were demonstrated in this report however in principle a spatially variable heat generation should be achievable with the initialisation of matrix of grid point heat generation values.

7. Future works

This project has only skimmed the surface of the possibilities for simulation of simple shaped structures. The objectives which were not achieved during the course of this project would all be within practical reach given more time. Further to this, the merging of mixed boundary conditions, annular geometry, internal heat generation and time dependency for a more comprehensive thermal profile. The implicit Jacobi method applied is very inefficient and has many more advanced/efficient alternatives at a minimum, it should be exchanged for Gauss-Seidel as the convergence speedup gained would outweigh the fractional time it would take to apply.

Modern batteries are made up of layers of different materials wrapped to form a spiral roll shape, the application of non-uniform material properties which vary spatially would also be a necessary step for useful thermodynamic battery simulation. A simplified first step would be to model several concentric annular shapes in contact with one another sharing boundaries at the intersections.

An interesting avenue for future work would be an implementation of the combined method which can remove the stability criteria associated with use of explicit time dependant solutions. This allows for longer and more usable simulation times and scenarios.

There are also a number of improvements that could be implemented to the current codebase. One of the most significant runtime costs incurred during testing was the *Parallel_Print_To_File( )* function. This will vary depending on a number of factors and it is no surprise that writing to a file is time intensive however the need to use collective calls to open and close the file appears to contribute to the problem. It would be beneficial to investigate alternatives for improvement further.

Explicit memory alinement was not implemented when allocating memory; this would make the code less portable but better suited to the cluster it is tested on. The code could also be made more user friendly, with greater flexibility at runtime. The ability to input more user specified parameters as opposed to the current hardcoded boundary conditions and grid dimensions which require file editing would make setting up and repeating simulations much easier.

8. Bibliography

[1] S. Petrovic, in *Battery Technology Crash Course*, Happy Valley, Oregan: Springer, 2020, pp. 95-98.

[2] Tesla, "www.tesla.com," 22 09 20. [Online]. Available: https://www.tesla.com/en_ie/2020shareholdermeeting. [Accessed 18 06 21].

[3] M. N. Ozisik, Heat Conduction, 2nd ed., Toronto: John Wiley & Sons, Inc, 1993.

[4] G. H. G. Wellein, Introduction to High Performance Computing for Scientists and Engineers, Miami: Taylor & Francis Group, LLC, 2011, pp. 115-140.

[5] jtang, "Trinnity College Dublin," Trinity College Dublin, 2011. [Online]. Available: https://www.tchpc.tcd.ie/resources/clusters/kelvin. [Accessed 15 09 21].

[6] "MPI_Dims_create(3) man page," The Open MPI Project, 20 03 2020. [Online]. Available: https://www.open-mpi.org.

[7] P. A. Ron, "Lecture 14: Linear Algebra," University of Wisconsin-Madison, Madison, 2010.

[8] V. Rajaraman, "IEEE Standard for Floating Point Numbers," *Resonance,* pp. 11-14, 2016.

[9] A. Mahmoodpoor, Director, *Analytical Solution of 2D Laplace equation.* [Film]. Russia.2020.

[10] R. Hugo, Director, *Heat Transfer L10 p1 - Solutions to 2D Heat Equation.* [Film]. America: Ron Hugo, 2016.

[11] t. e. al, "18650 cs. 21700 Li-ion cells - A direct comparison of electrochemical, thermal, and geometrical properties," *Journal of Power Sources,* vol. 472, no. 228614, pp. 5-6, 2020.

[12] M. C.Wendl, "One-Dimensional Steady Conduction," in *Fundamentals of Heat Transfer Theory and Applications*, Saint Louis , Washington University, 2005, pp. 15-27.

[13] X. Li, "Vetenskap Och Konst," PDC Centre for High Performance Compution, 09 11 2018. [Online]. Available: https://www.kth.se. [Accessed 09 2021].

9. Appendix

9.1 Code Files

Table 9.1-1: List of Appended files

Caption No.	Filename	File Format
file 9.1-1: C code for solving 2D Rectangular Stead State with Fixed Boundary	2D_Rect_Fixed_Boundary_SS	.c
file 9.1-2: C code of functions performing Sweep, Exchange, and Convergence	jacobi_Itteration	.c
file 9.1-3: Header file for including Jacobi_itteration.c function declarations	jacobi_Itteration	.h
file 9.1-4: Make file for quick compilation of executables from written c files	Makefile	
file 9.1-5: Grid management file for memory allocation, serial printing and initialization	memory_alloc	.c
file 9.1-6: Header file for memory_alloc.c	memory_alloc	.h
file 9.1-7: A file written for testing various memory_alloc.c functions	memory_alloc_test	.c
file 9.1-8: A file for similtanious printing of processor grids to a common binary file	Parallel_write_to_file	.c
file 9.1-9: Header file for Parallel_write_to_file.c	Parallel_write_to_file	.h
file 9.1-10: An instructional file on the compilation and running of executables	README	.md
file 9.1-11: A bash script written to run jobs on the cluster	Sbatch	.sh
file 9.1-12: A file written to test printing, initialisation of dynamically allocated grids on many processors	Dynamic_write_to_file	.c
file 9.1-13: A file written to test the decomposition of processors over the grid axis	decomp_rect_test	.c
file 9.1-14: A file written and used to test the decomposition of 2d grids	decomp2test	.c
file 9.1-15: A file for performing decomposition of both the processors over the axis and processors over the grid	decom2d	.c
file 9.1-16: Header file for decomp2d.c	decomp2d	.h
file 9.1-17: A file used for testing the Parallel print function	Parallel_write_to_file_test	.c
file 9.1-18: A project diary file containing dated entries of works done, problems and queries to follow up on	Dear_Diary_Thesis	.txt
file 9.1-19: A function for converting binary file data to structured .txt	Binary_to_decimal_txt	.py
file 9.1-20: C code for running 1D Transient conduction through a 2D annulus shape with fixed boundary conditions	2D_Annulus_Transient_Fixed_Boundary	.c
file 9.1-21: C code for solving 1D heat conduction through a 2D annulus shape with fixed boundary conditions	2D_Annulus_Fixed_Boundary	.c
file 9.1-22: C code for solving 1D heat conduction through a 2D annulus shape with convective boundaries	2D_Annulus_Convective	.c
file 9.1-23: A Mathlab file for generating an analytical square grid solution, importing a numerical .txt and generating plots to compare both	Analytical_poisson	.m
file 9.1-24: A Mathlab file for generating an analytical annular grid solution, importing a numerical .txt and generating plots to compare both	Cylindrical_plot	.m
file 9.1-25: A Mathlab file for comparing two numerical results run over different numbers of processors to ensure consistent results	Test_multiple_procs_match	.m