# Multigrid Improvements to CITCOM (a dCSE project)

Dr Sarfraz Ahmad Nadeem
NAG Ltd.

Peter House, Oxford Street
Manchester M1 5AN
UK

UK Corporate Headquarters:
Wilkinson House, Jordan Hill Road
Oxford, OX2 8DR
UK

# Acknowledgements

My gratitude to the project PI, Dr Jeroen van-Hunen, for his help, support, discussion and availability for meetings including a few on very short notice and visits to NAG office in Manchester.

Thanks to Group Leader, Dr Edward Smyth, and dCSE Manager, Dr Phil Ridley, for answering questions and offering guidance whenever required.

Thanks to NAG staff, particularly those based in NAG office in Manchester, for helping to create an excellent work environment, helpful discussions and caring for each other.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 About the CITCOM dCSE Project

This dCSE project was proposed by Dr Jeroen van Hunen, Department of Earth Sciences, University of Durham along with Dr Charles E. Augarde, School of Engineering, University of Durham. It is targeted at the multigrid (MG) improvements to the CITCOM package, a parallel finite element code designed to solve thermal convection problems relevant to geodynamics. The code is written in `C` and parallelisation is based on the `MPI` library.

### 1.1.1 Project Duration

This project was granted 12 months full time effort for one person to work on a full time basis which, as per a flexible working policy, translates to 15 months time on an 80% basis. It started on January 1, 2009 with a scheduled end date of March 31, 2010 and consisted of three phases, each one dedicated to one or more specific tasks defined in the project proposal and approved as appropriate. A breakdown of these tasks can be described as:

1. Initial project study;
2. Multigrid cycles and smoothers;
3. Mesh refinement.

### 1.1.2 Project Plan

The aim of this dCSE project is to improve parallel performance and scalability by improving the convergence rate. This would be achieved by implementing/improving algorithmic and programming enhancements for the existing multigrid cycles (where only the V-cycle was initially implemented) with the aim being to implement W-cycle and F-cycle methods, along with mesh refinement and communication amongst the processes. As per proposal, the project is divided into three phases as described in section 1.1.1. After the initial project study in the first phase, during the next two phases the potential candidates for improvements are:

- Phase 2
  (a) **MG cycles**
  By improving the currently implemented V-cycle and implementing W-cycle and F-cycle.
  (b) **Smoothing**
  By replacing the existing Jacobi solver with a Gauss-Seidel solver that will implement a black/red inter-processes communication scheme for the boundary nodes (but see further details on this in Chapter 2).
- Phase 3
  (c) **Mesh refinement**
  By implementing local mesh refinement near the high viscosity gradients.
  (d) **Improved prolongation and restriction**
  Interpolation and transfer of information between MG levels. However, this requires a level of further study/investigation to assess the feasibility and suitability of CITCOM for this implementation.

Further details of these tasks along with implementation and performance improvement results are discussed in the following chapters in detail.

## 1.2 CITCOM Background

This two dimensional / three dimensional Cartesian version of CITCOM was originally written by Louis Moresi whereas Shijie Zhong later parallelised

and improved the code by implementing the full multigrid (MG) algorithm together with a consistent projection scheme, along with the re-ordering of the stiffness matrix and its matrix operations such that only half of the stiffness matrix is stored.

# Chapter 2

# Initial Project Study

As per the revised work plan, the first phase is of four months duration with a start date of January 1, 2009 and end date of April 30, 2009. This also included an interim report with a description of this phase along with planned code modifications and tests to be carried out in the next phase. There was no implementation work planned for this phase as it was to gain CITCOM related knowledge which was necessary in order to be able to work with the source code and then perform modifications and further implementations in the following two phases.
Four months duration of the first phase for the "Initial Project Study" is necessary for reasons explained in the subsequent paragraphs.

## 2.1 Background

This CITCOM package is PI's own copy of the Cartesian version of CITCOM source code and originated from the time he was working as postdoctoral researcher at the University of Colorado at Boulder, USA within Department of Physics during 2002–2004. In contrast to the Spherical version of the CITCOM which is well maintained and documented and is available at the Computational Infrastructure for Geodynamics web site [5], this Cartesian version has less documentation for the new users/developers to get kick started. Also, the source code is commented relatively sparsely otherwise our effort to get documentation extracted from the source code using Doxygen [8] would have been reasonably rewarding.

## 2.2 Learning Curve

Despite all the difficulties related to understanding the source code due to lack of documentation and sparsity of comments within the source code, getting to grips with the code was achieved by a number of ways including, but not limited to:

- Browsing through the source code to read the source code itself and sparse comments.
- Making use of Doxygen to extract as much as we could, at least "Call" graphs and "Called by" graphs were of help.
- Making use of etrace package [6] to learn function call tree.
- Meetings and discussions with CITCOM project PI, where his *little–yellow–magic–notebook* proved a very valuable asset.
- Googling.

As a result of our combined efforts, we became close to the point where with some confidence we could say that we are ready for the next phase of the project and to undertake code development work.
Efforts to learn more about the CITCOM package and implementation continue throughout this project. Getting hands on with the CITCOM packages, modifying existing source code, adding new functions, etc. is all useful.

### 2.2.1 Learning CITCOM

This CITCOM package is built on a structured finite element mesh which is made up of rectangular elements in two dimensions (2D) and brick elements in three dimensions (3D). These elements reduce to a square in 2D and a cube in 3D respectively. Given that an appropriate number of elements are used in each dimension with respect to the length of the corresponding dimension and as the size of each element along any axis is dependent of the number of elements and the overall length of the mesh along that direction. Such elements can be represented by figure 2.1 in the two dimensional case and by figure 2.2 in the three dimensional case respectively.
Our main focus here are three dimensional brick elements which for our application are more appropriate for the study of the convection of the earth's

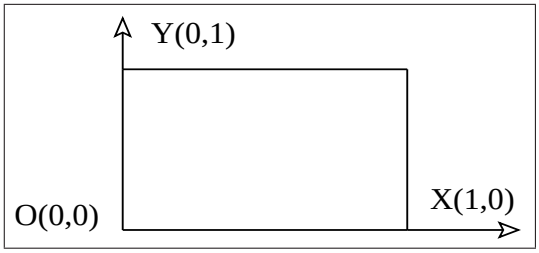


Figure 2.1: A standard rectangular element in 2D

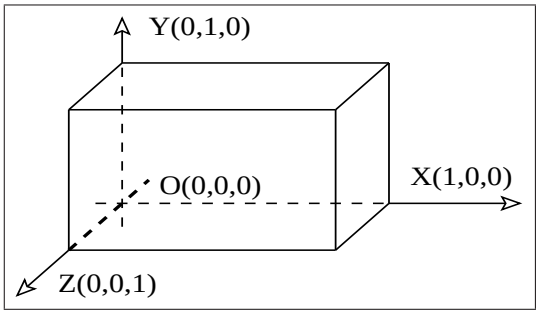


Figure 2.2: A standard brick element in 3D

mantle. However, to make the learning systematic and smooth, we would continue with the description in two dimensions as well as in three dimensions. It is interesting to learn that structured mesh elements used by CITCOM, in contrast to the standard Cartesian coordinate system `(x,y)` in 2D and `(x,y,z)` in 3D, have a coordinates system `(x,z)` in 2D and `(x,z,y)` in 3D as shown in figure 2.3 for the two dimensional rectangular element and in figure 2.4 for the three dimensional brick element respectively.
This fact is further highlighted in figure 2.5 for the two dimensional rectangular element and in figure 2.6 for the three dimensional brick element along with given coordinates for a unit element where it is also reflected that the values of `z-coordinates` along the downward `z-axis` are taken `positive`. Local node numbering for each element is `counter-clockwise` starting at the origin `(0,0)` in 2D as shown in figure 2.5. This is further elaborated in the 3D case where the local node numbering for an element starts at the origin `(0,0,0)`, first on the front-face of the element followed by on the back-face of the element as shown in figure 2.6. This local node numbering

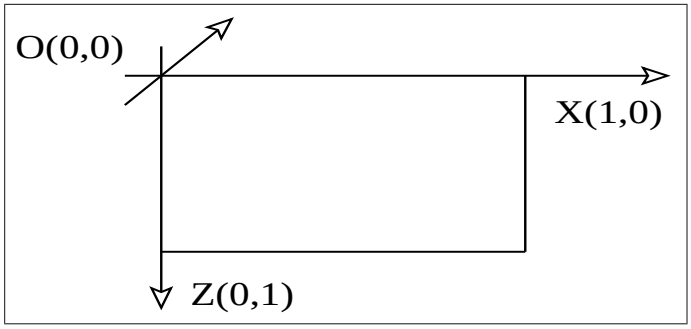


Figure 2.3: A CITCOM rectangular element in 2D

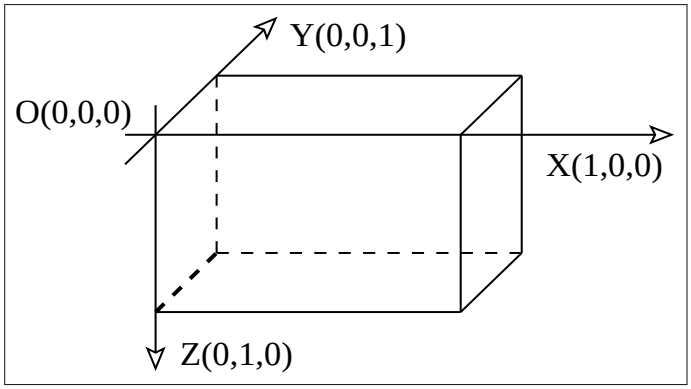


Figure 2.4: A CITCOM brick element in 3D

runs from `1` to `4` in 2D and from `1` to `8` in 3D with the `0` (zero) position leaving un–used, in contrast to the `C` language standard (the language of the CITCOM package) where everything begins at `0`. This is a major feature of CITCOM that almost everywhere the `0` (zero) position is un–used and an extra position is allocated where needed in lieu of this un–used position. However, there are a few exceptions.

There is another point which could be of interest. In most cases, instead of one extra position in lieu of un–used `0` (zero) position, two extra positions are defined/allocated.

Also, no element on the mesh surface is considered as a surface element except those which are on the earth's surface considering the mapping between mesh and earth. In other words, only those elements which touch the `X-axis` in two

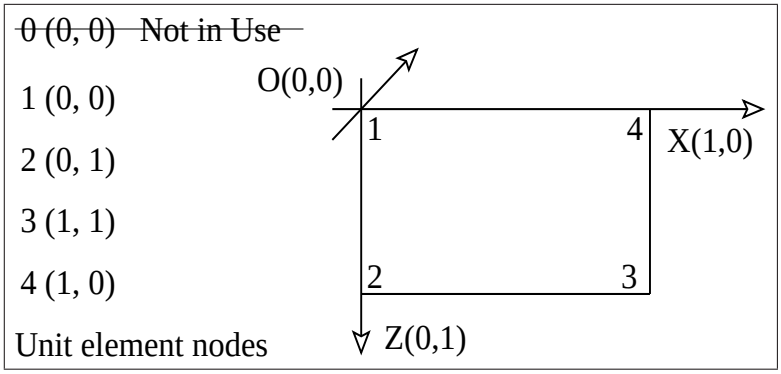


Figure 2.5: A CITCOM rectangular element and local nodes relationship

dimensions and `XOY-plane` in three dimensions are considered to be surface elements. Similarly, only nodes which are on the `X-axis` in two dimensions and `XOY-plane` in three dimensions are considered to be surface nodes.
To elaborate further on this, in the two dimensional case as shown in figure 2.7, `element 4` is not a surface element whereas `element 7` is a surface element. Also, `element 4` has no surface node whereas `element 7` has two surface nodes which are `10 & 13`. To emphasise this, any element with surface nodes will have exactly two nodes on the surface irrespective of element position within the mesh.
Similarly, in the three dimensional case as shown in figure 2.8, `element 4` is not a surface element whereas `element 15` is a surface element. Also, `element 4` has no surface node whereas `element 15` has four surface nodes which are `40, 25, 28 & 43`. To emphasise this, any element with surface nodes have exactly four nodes on the surface irrespective of element position in the mesh.
It should be noted that the previous paragraphs which describe surface elements in a mesh are from the "numerical" surface of a finite element mesh perspective rather then from the "earth science" perspective. In the later case, the scenario is obvious and well understood. Hence this might just be a nomenclature issue but it is included here for the sake of clarity.
Mapping of local node numbering for any given element to the corresponding global node numbering is another important relationship for any finite element mesh. This helps to establish mesh connectivity and the relationship between the nodal coordinates for any given node in the mesh. Such a

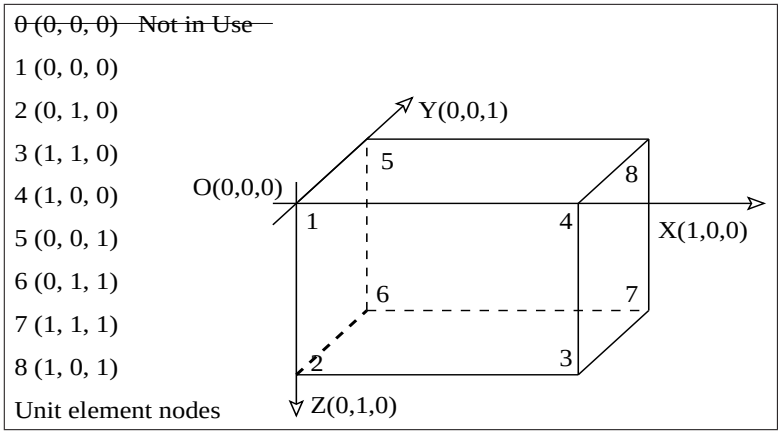


Figure 2.6: A CITCOM brick element and local nodes relationship

connectivity relationship is depicted in figure 2.7 for element `4` and element `7` for the two dimensional case and in figure 2.8 for element `4` and element `15` for the three dimensional case. Having established these relationships for a finite element mesh, it is not hard to compute the required parameters of the mesh such as global nodes for other given parameters such as elements and vice versa. For example, for any given element number, it is not hard to find a global node number corresponding to a local node number for that element and hence coordinates for the global node. Conversely, for any given global node number it can easily be established just how many elements in the mesh this particular node belongs to. In all mesh elements, local nodes, global nodes, node coordinates, relationship, etc. extra complexity is introduced by the `levels` of mesh and all of these entities exist at each level of the mesh. Each level in the mesh represents a certain depth of when an element is refined or sub–divided, for example, if a certain element in the mesh is refined or sub–divided three times, all the resulting elements belong to the third level of the mesh. These `levels` of the finite element mesh play an important role in the multigrid algorithms implemented (or to be implemented) in the CITCOM package. Rather, it is more appropriate to say that these `levels` in the finite element mesh provides the basis for multigrid algorithms.

Let us come back to the mesh element-node relationship. We notice that all numbering for elements, local nodes and global nodes starts at `1` rather than `0` (zero). This element numbering and global node numbering begins

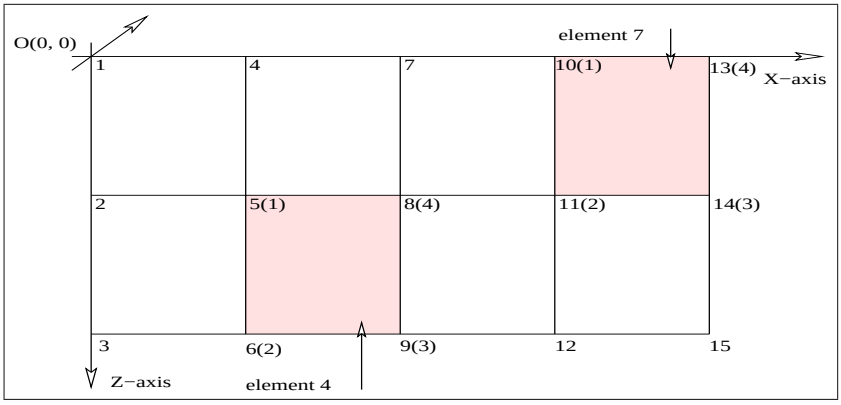


Figure 2.7: A sample CITCOM mesh of 8 (4x2) rectangular elements

at the origin of the mesh, and gets incremented along the `z-axis`, followed by the `x-axis` and in the 3D case followed by the `y-axis`. This incremental order for the element and global node numbering plays an important role in calculating any entity, whether it is an element–node relationship or the computation of velocity, pressure, viscosity, etc.
In addition to this order of element and global node numbering in the mesh, a number of `C` data structures defined in the `header` files are of great importance in implementing algorithms in CITCOM but probably it is not sensible to go into those details here.

## 2.3 Precautions

There is no restriction on the number of processes any CITCOM job can run for or any specific sequence of the number of processes such as `2, 4, 8, ...`. However, there are some very basic rules which must be met for any CITCOM job to be completed successfully. In this regard, a few precautions, as described below, must be taken.

- The number of `mgunit` in the `(x, z, y)-direction` must be at least twice the number of processes in the `(x, z, y)-direction` respectively. These `mgunit<x,z,y>` define the number of elements along the `(x, z, y)-direction` for the corresponding structured finite element mesh at the coarsest level. Thus, the exact number of elements and

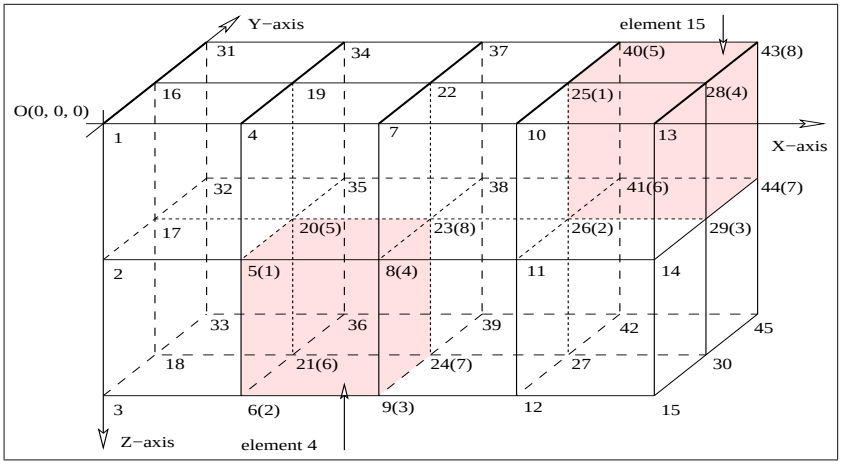


Figure 2.8: A sample CITCOM mesh of 16 (4x2x2) brick elements

the number of global nodes in the `(x, z, y)-direction` for the given mesh level are defined as:

- `number of elements = mgunit<x, z, y> *` $2^{(level-1)}$
- `number of nodes = mgunit<x, z, y> *` $2^{(level-1)} + 1$

This helps determine the number of global nodes along the `(x, z, y)-directions` to be specified in the CITCOM input file.

The description for the three dimensional case as given above is also valid for the two dimensional case.

- Any number of processes can be defined in the CITCOM input file as long as they are consistent with the other predefined parameters, except the number of processes along the `z-direction` which must be `1` (one).

## 2.4 Miscellaneous

This section includes the work which may not necessarily contribute towards the improvement of the performance of CITCOM but might improve the CITCOM code in order to make it more user friendly and easy to understand.

The first of these such additions is described below, others could be added later as and when implemented.

### 2.4.1 `data_directory()`

The CITCOM code reads in all of its required parameters from an input file which is read by CITCOM as first and only input argument at the command-line. In this input file, a user is required to provide an output directory and the base name for the output files in the form of `<dirname>/<basefilename>` against the input keyword `datafile`. (**Note:** The user is free to add new keywords and/or change existing keywords. Precautions must be taken as this may trigger changes to the code itself unnecessarily.)
In order to run CITCOM, a user is required to create an empty output directory before submitting the job, or more appropriately before the start of execution, within the directory from where the CITCOM job will run. If the user forgets to create the directory and the job starts executing, it is certain that the job will crash. However, if the directory was specified in the input file as existing from a previous run, every thing will be overwritten unless the base file name is different for the current run.
Addition of the function `data_directory()` has eliminated this unpleasant possibility. The iser is no longer required to create a directory as specified in the input file. It will be created at run time. If the directory with the same name as specified in the input file exists, it will be renamed to a directory, the name of which is generated by attaching the current date and time (as explained in section 2.4.2) with current process identity as an extention to the existing name in the form `<directory>.<yyyymmdd>-<hhmmss>.<pid>` to make it unique. Then, the directory as specified in the input file is created by the `root or 0` process and it is assured that no process is allowed to write to any of the output files until this new directory is created.
This would ensure that no job ends up wasting compute resources and no results from a previous run are over written due to minor negligence at the user end.

### 2.4.2 `date_and_time()`

This function utilises C' built in time struct (`struct tm`) and returns the current date and time in the format `<yyyymmdd>-<hhmmss>` as required by

the function `date_directory()` (see section 2.4.1) to make the output data directory name unique.

## 2.5 Planned Implementation and Tests

### 2.5.1 Implementation

During the second phase of this dCSE project, the implementation tasks as defined in the project proposal are to be implemented. These tasks are outlined very briefly as follows:

- Study of the currently implemented multigrid V-cycle.
- Implementation of multigrid W-cycle.
- Implementation of multigrid F-cycle.

### 2.5.2 Testing

Once the implementation of these proposed multigrid cycles is complete, we would go through testing all three multigrid cycle implementations against a comparatively smaller to medium size test problem. Hence the tasks are:

- Test multigrid V-cycle for a smaller to medium size test problem(s).
- Test multigrid W-cycle for a smaller to medium size test problem(s).
- Test multigrid F-cycle for a smaller to medium size test problem(s).

### 2.5.3 Optimisation

All the test results will then be compared against standard benchmarks for the selected test problem(s). After verifying the accuracy of the test results, the implementation of multigrid cycles will go through an optimisation phase. This will require completion of the following tasks:

- Optimise multigrid V-cycle implementation.
- Optimise multigrid W-cycle implementation.
- Optimise multigrid F-cycle implementation.

At this point it would be especially interesting to see/note if the optimisation would be problem-dependent, and if so, what characteristics determine the optimal conditions for these cycles.

### 2.5.4 Further Testing

Next comes the testing of optimised multigrid cycles for test problem(s) ranging from medium to large and possibly very large size and subject to the feasibility of any such test problem in the context of CITCOM capability and the limitations of the problem itself.

- Test multigrid V-cycle for medium to large size test problem(s).
- Test multigrid W-cycle for medium to large size test problem(s).
- Test multigrid F-cycle for medium to large size test problem(s).

After satisfactory completion of the test runs and verification of results, other aspects of the CITCOM package would be considered for optimisation as appropriate. This should lead to the next and final test runs for a number of benchmark test problems by the end of which we are expecting to have reasonable performance results available for the report on the end of the second phase.

## 2.6 Conclusions

On completion of the first phase of the project within prescribed timescale, multigrid cycles implementation will be undertaken in the next phase. On the basis of lessons learnt, it has been understood and recognised mutually that there should have been more time dedicated towards the learning and better understanding of this package. In line with this need, time will be spared to learn more about the CITCOM source code during the next two phases on the basis of as and when feasible. At the same time, it been agreed that we will continue to document and improve the existing documentation both in the form of source code commentary and this report.

# Chapter 3

# Multigrids

In order to solve a system of linear equations, typically one of the following two classes of solvers is used.

1. Direct methods
2. Iterative methods

Direct methods originate from Gaussian elimination, they have a limited application field and determine the solution at machine precision accuracy with optimal performance of $O(n^2 log(n))$ [7] where $n$ is the number of equations/unknowns in the linear system.
Iterative methods have wider field of applications with Jacobi, Gauss-Seidel and Conjugate Gradient being the most popular amongst others. However, attempts to overcome the unconvincing time efficiency of these methods resulted in an extension to iterative methods called multigrid methods.

## 3.1 Multigrid Methods

Multigrid (MG) methods are a branch of numerical analysis and comprise of algorithms for systems of linear equations typically obtained after the discretization of partial differential equations. Elliptic partial differential equations are the prime candidates for the multigrid algorithms/applications. These algorithms employ a hierarchy of discretizations at more than one level of underlying grid. The purpose of these algorithms is that at multiple grid levels, called multigrids, they accelerate the rate of convergence. This

acceleration is achieved by employing a number of steps in a specific order and includes:

1. Relaxation/Correction of the global solution by solving a coarse grid problem
   *Helps reduce high frequency errors by employing Gauss-Seidel like iterative methods, beginning at the finest grid and gradually moving down to the coarsest grid where it becomes a coarse grid problem.*

2. Restriction/Projection of the residual vector of the current approximation from a fine grid to a coarse grid
   *Restricts/Projects down the residual vector to a coarse grid by employing some form of averaging, for example.*

3. Prolongation/Interpolation of the approximation to a residual vector from a coarse grid to a fine grid
   *Prolongates/Interpolates an approximation to the residual (error) at the coarse grid to a fine grid and is used for correction there.*

These three steps are building blocks or components of any multigrid algorithm and each of these component is briefly described below.

### 3.1.1 Relaxation/Correction

It has been established [4] that many standard iterative methods possess the smoothing property. This property makes these methods very effective at eliminating the high frequency or oscillatory components of the error and leaves the low frequency or smooth (less oscillatory) components relatively unchanged. These iterative methods can be further improved to handle error components of all types equally effectively. Towards this end, relaxation schemes can be improved by using a good initial guess. A well known technique to obtain a good initial guess is to perform a few preliminary iterations on a coarse grid. As there are fewer unknowns on a coarse grid, relaxation is less expensive and the rate of convergence on a coarse grid is marginally better as well [4]. Further, a coarse grid solver also takes advantage of the fact that smooth or low frequency components of error on a fine grid take the form of oscillatory or high frequency components of error on a relatively coarse grid. In other words, when a relaxation scheme on a fine grid begins to stall due to smooth error modes, it is the time to move to the coarse grid

where these error modes appear more oscillatory and the relaxation scheme becomes more effective. A combined effect of this, along with other MG components, results in a substantial improvement to the rate of convergence of the global problem.
In this context, consider the following steps:

- Relax on fine grid for an approximation to solution.
- Compute residual vector of the current approximation at fine grid.
- Transfer this computed residual vector to the coarse grid by means of a restriction operator.
- Relax on residual equation on coarse grid to get an approximation to error.
- Transfer this error approximation to fine grid by interpolation operator.
- Correct the approximate solution obtained on fine grid with error approximation at coarse grid.

This procedure outlines the correction scheme. Theoretically, relaxation is employed on a fine grid to approximate the solution until convergence stagnates or deteriorates. In practice, relaxation is applied only for a few steps instead of waiting for convergence to stagnate or deteriorate. This helps optimise MG schemes. However, the optimal choice of the number of steps that relaxation is applied may vary for different problems. Relaxation is then applied to the residual equation on the coarse grid to get the approximation to the error itself which is then used to correct the solution approximation already obtained on fine grid. This fine grid and coarse grid scenario when extended to more than at least two grid levels refers to multigrid and requires transferring the residual vector to multiple levels in the grid hierarchy, from the finest grid to the coarsest grid via intermediate grids. The same procedure is required to transfer approximations to the residual (error) but in a reverse order, from coarsest grid to finest grid. These inter-grid transfer procedures are referred as restriction/projection and prolongation/interpolation respectively.
As improvements to prolongation and restriction operators (along with the mesh refinement) are topics of study planned for phase 3 (the next phase) within this project, these are not discussed here except the description of the basic functionality of each of these operators for the sake of completeness.

### 3.1.2 Prolongation/Interpolation

Prolongation/Interpolation, in the multigrid context, is an operator used to transfer error approximation from coarse grid to fine grid. It can be of any order but even the simplest, linear interpolation for example, works quite effectively.

### 3.1.3 Restriction/Projection

As the interpolation operator is used to transfer the error approximation from a coarse grid to a fine grid, an operator with the reverse or counter effect that transfers the residual vector from fine grid to coarse grid is called a restriction/projection operator. The most obvious example for such an operator is the so called injection whereas a more elaborate alternative operator is the full weighting or weighted average.

## 3.2 Multigrid Cycles

Multigrid schemes available within CITCOM at the end of this phase (phase 2) are described in this section. The first two schemes implemented in CITCOM are known as multigrid V-cycle and W-cycle schemes. The next two schemes are known as Full Multigrid or FMG schemes, these can also be referred to as F-cycle schemes. Based on V-cycle and W-cycle within the FMG scheme, these are referred as FMG(V) and FMG(W) schemes respectively.

### 3.2.1 V-cycle

Starting at the finest grid, the V-cycle scheme involves a number of relaxation steps followed by computation of a residual vector which is than transferred to the next coarser grid with the help of a restriction operator. This procedure is repeated until the coarsest grid with a relatively few grid points (in comparison to the finest grid) is reached. This scheme then works its way back to the finest grid while employing the correction just computed to the next fine grid which is transferred by means of the prolongation operator. This procedure is repeated until the finest grid is reached. At this point one iteration of the V-cycle is completed. A graphic representation of a V-cycle iteration is shown in figure 3.1.

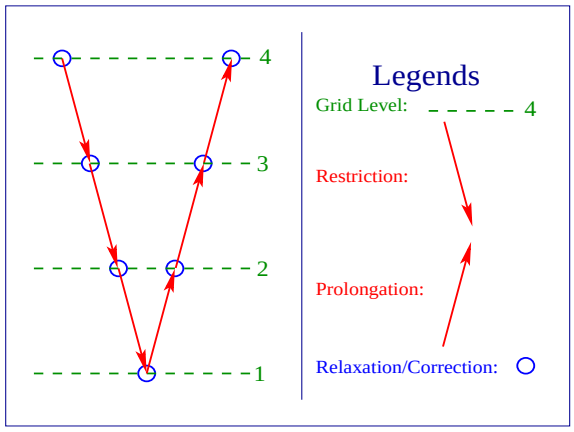


Figure 3.1: Multigrid V-cycle Scheme

After each completed iteration of the V-cycle scheme, convergence of the global problem is examined. If a pre-defined convergence criteria is satisfied, the procedure is terminated. If not, this V-cycle scheme is repeated until convergence is reached or otherwise the procedure is terminated.

### 3.2.2 W-cycle

The W-cycle scheme is similar to the V-cycle scheme as long as the key components of the scheme namely relaxation/correction, restriction and prolongation operations and their functionality is concerned. The only difference, in its simplest form, is that the convergence of the global problem is examined after the V-cycle iteration and this is repeated twice to make it a W-cycle scheme as shown in figure 3.2.

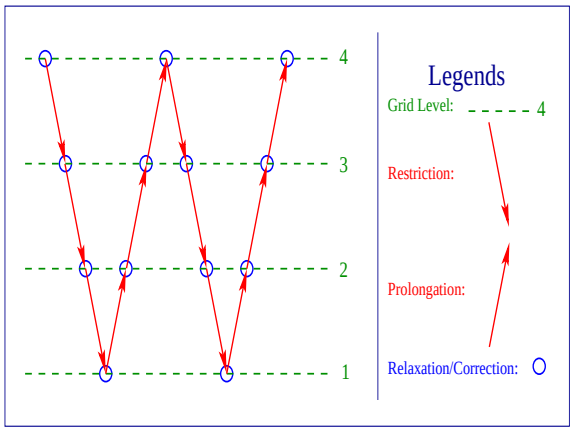


Figure 3.2: Multigrid W-cycle Scheme

### 3.2.3 FMG Cycles

Full Multigrid (FMG) schemes, FMG(V) based on V-cycle and FMG(W) based on W-cycle, work exactly is the same way as V-cycle and W-cycle schemes respectively. The difference lies in the iteration which includes all grid levels in the case of V-cycle and W-cycle schemes in contrast to the iteration just over two coarsest grid levels at the beginning and this extends itself to include the next fine grid level, one by one, until it reaches the finest grid. In other words, it starts a V-cycle (in the case of the FMG(V) scheme) over two coarsest levels and extends itself incrementally to all grid levels. Due to incremental inclusion of the next fine grid level into the V-cycle iteration that started over two coarse grid levels ago and eventually covered all grid levels (full grid), this scheme is referred as the FMG(V) scheme (also referred as F-cycle) as shown in figure 3.3. The FMG(W) scheme is not fundamentally different from the FMG(V) scheme except that it iterates like the W-cycle instead of the V-cycle as described in section 3.2.2.

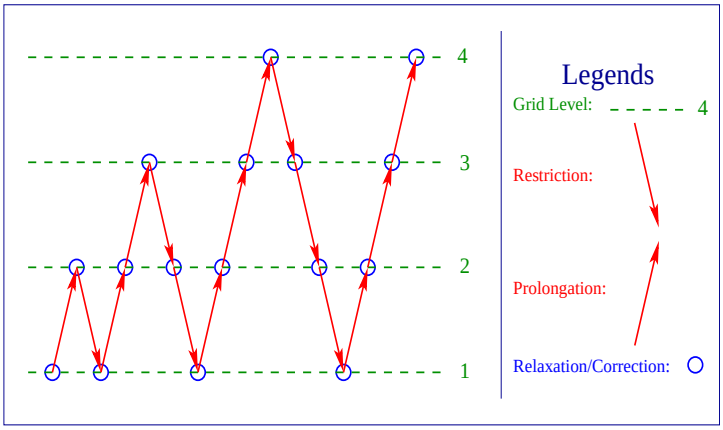


Figure 3.3: FMG(V)-cycle Scheme

### 3.2.4 Multigrid Schemes

To summarise, at the end of phase 2 of this project, following four multigrid schemes are available to solve a variety of problems.

1. Multigrid V-cycle scheme.
2. Multigrid W-cycle scheme.

3. FMG(V) scheme.

4. FMG(W) scheme.

## 3.3 Model Problem

The finite element based CITCOM code[11, 12, 15] is designed to solve fluid flow problems with temperature and composition dependent density and viscosity. The governing equations can be described by a set of conversation equations for mass, momentum, energy and composition as:

$$\nabla \cdot \overrightarrow{v} = 0 \tag{3.1}$$

$$\nabla\tau + \nabla p = \Delta\rho g\hat{z} \tag{3.2}$$

$$\frac{\partial T}{\partial t} + (\overrightarrow{v} \cdot \nabla)T = \nabla^2 T + H \tag{3.3}$$

$$\frac{\partial C}{\partial t} + (\overrightarrow{v} \cdot \nabla)C = 0 \tag{3.4}$$

In these equations, $\overrightarrow{v}$ represents the velocity field, $\tau$ the reduced stress field, $p$ the reduced pressure field, $\Delta\rho$ is the relative density dependent on temperature $T$ and composition $C$, $g$ is the gravitational acceleration, $\hat{z}$ is the unit vector in vertical direction, $t$ is the time and $H$ is the heat production rate. Iterative MG solution methods are used to solve the Stokes equations 3.1 and 3.2, temperature equation 3.3 is solved using a simple explicit forward integration scheme and composition equation 3.4 is solved using a tracer method. Equations 3.1 and 3.2 in their discrete form can be written as:

$$Au + Bp = f \tag{3.5}$$

$$B^T u = 0 \tag{3.6}$$

This system of equations is solved using the Uzawa iteration scheme where at each MG level equation 3.5 is solved iteratively with a Gauss-Seidel method except at the coarsest grid where it is solved using Conjugate Gradient method and equation 3.6 is applied as a constraint.

## 3.4 Test Problems

For the governing equations (3.1–3.4) representing the model problem, four representative test problems are considered in this section. This selection of test problems, in two and three dimensions, covers a variety of geodynamical problems related to the Earth's mantle and lithosphere such as:

- Mantle convection
- Subduction zones
- Mantle plumes

These four representative test problems are described as:

1. This "simple 2D test problem" represents time dependent (Rayleigh–Benard) convection with constant viscosity in a square box ($1/h = 1$) with temperature fixed to zero at the top, to $\Delta T$ at the bottom and no internal heat source. However, when run for long enough, that is, thousands of timesteps on a reasonably coarse mesh, and even much more timesteps on a finer mesh, this will produce a steady-state solution if $Ra < 10^6$. It reflects symmetry at the side walls (i.e. $\partial_x T = 0$) and zero shear stress on all boundaries of the square box. The Rayleigh number is $Ra = \alpha g \Delta T h^3 / \kappa v = 10^4$ [2] where $h$ is of the order of hundreds to thousands of kilometres for mantle convection.

2. Time dependent (Rayleigh–Benard) convection with constant viscosity in a unit cube ($1/h = 1$) with temperature fixed to zero at the top, to $\Delta T$ at the bottom and no internal heat source. However, when run for long enough, that is, thousands of timesteps on a reasonably coarse mesh, and even much more timesteps on a finer mesh, this will produce a steady-state solution if $Ra < 10^6$. It reflects symmetry at the side walls along the x-direction (i.e. $\partial_x T = 0$) and zero shear stress on all boundaries of the unit cube. The Rayleigh number is $Ra = \alpha g \Delta T h^3 / \kappa v = 10^4$ [2] where $h$ is of the order of hundreds to thousands of kilometre for mantle convection. This 3D test problem is an extension of the above simple 2D test problem where a square box is stretched along the y-direction to make it a unit cube. It is referred to as a "simple 3D test problem".

3. This "complex 2D test problem" is an extension of the simple 2D test problem and it represents the important geodynamical process of subduction (where two lithospheres, or plates meet, and one subducts beneath the other). This extension deals with large and sharp viscosity contrast. To model this, a temperature-dependent viscosity (with cold material being strong) of the form $\eta = A^{(E^*/RT)}$ with $E^*$ the activation energy of mantle material (around 400 kJ/mol), R is the gas constant, and T the absolute temperature. This viscosity description increases by many (30 or so) orders of magnitude when $T$ goes from a mantle temperature of $1350^0C$ down to $0^0C$ near the surface. As such large viscosity contrast is impractical to model correctly, the maximum viscosity is cut off to $10^3 - 10^5$ times the hot mantle viscosity (the latter being taken as the reference viscosity). The larger the $\eta$-contrast in the calculation, the more difficult for the code to find convergence for the Stokes equations 3.1 and 3.2. In addition, decoupling of the two converging plates requires a thin, low viscosity zone. In this test problem, this low-viscosity zone is set to have a width of tens of kilometres (compared to the full box size of thousands of kilometres), and has a viscosity similar to that of the hot mantle (even though it sits within the cold, much stronger lithosphere). This low-viscosity zone and the surrounding lithosphere form a sharp viscosity contrast.

4. This "complex 3D test problem" has many of the same features as the complex 2D test problem. It represents a subduction setting, but one that has ceased to converge (a scenario that occurs when two continents collide, and neither of them is able to subduct). In this case the previously subducted lithosphere hangs vertically in the mantle, and eventually breaks off or diffuses away in the hot mantle. This setup is now modelled in full 3D (which is computationally more demanding), but doesn't require the weak decoupling zone so that convergence is expected to be somewhat better.

## 3.5 Results

This section consists of results for the test problems as described in section 3.4. Each of these test problems is solved using all four multigrid schemes available at the end of the second phase of this project. The solution times

are presented in seconds for the MG schemes and the number of MPI processes used to get solution in that particular case. The times for MG schemes and the corresponding number of MPI processes are then presented graphically followed by plot in log scale to reflect parallel performance and scaling. Further interpretation of the results is provided with the help of charts representing temperature and velocity fields. These charts are plotted in the xz-plane for temperature and velocity fields for test problems in 2D. The same is repeated in the xz-plane and yz-plane for the temperature field and in the xy-plane and xz-plane for the velocity field, for test problems in 3D.
Solutions for test problems in 2D are obtained for 2, 4, 8, 16, 32 and 64 MPI processes. However, the quad core configuration of HECToR [9], that is, four cores per node have some restrictions on fitting the data to the available memory during run time. The available memory configuration of 8 GB per node or 2 GB per core on HECToR is not sufficient in the case of 2 and 4 MPI process jobs for the 2D test problems under consideration due to the increased size of sub problem per MPI process. This limits the test runs for 2 MPI process jobs to use one core per node and test runs for 4 MPI processes jobs to use two cores per node only. In all other cases four cores per node are used for test problems in 2D. Similarly, solutions for test problems in 3D are obtained for 32, 64, 128 and 256 MPI processes. Again, the memory configuration of 8 GB per node or 2 GB per core on HECToR is not sufficient in case of 32 and 64 MPI process jobs for the 3D test problems under consideration due to the increased size of sub problem per MPI process. This limits the test runs for 32 MPI processes jobs to use one core per node and test runs for 64 MPI processes jobs to use two cores per node only. In all other cases four cores per node are used for test problems in 3D.
This way, jobs using one core per node become four times as expensive and jobs using two cores per node become two times expensive as compared with jobs using four cores per node if the run is for the same length of time. This encourages us to make better use of compute resources by employing more MPI processes per node in order to take full advantage of the facility.
Before discussion about the individual test problems and results, it should be noted that first test problems in 2D and 3D are considered “simple” in comparison to the following two test problems which are considered relatively “complex”. This comparison is based on the physical nature and the corresponding computational efforts required for the selected set of test problems. This selection of test problems is by choice and found to be helpful in identifying strengths and weaknesses of the MG schemes.

### 3.5.1 Simple 2D Test Problem

The time taken by each MG scheme to complete 100 timesteps in the case of the simple 2D test problem is presented in table 3.1. These times are plotted against the number of MPI processes in figure 3.4 (left) which shows that the V-cycle scheme is performing better when compared to the other three MG schemes for MPI processes between 2 and 32. The W-cycle scheme becomes better for 64 MPI processes. However, it is to be noted that performance of FMG(V) and W-cycle schemes are very close to each other.

| Number of Processes | Time (in seconds) | | | |
|---|---|---|---|---|
| | V-cycle | W-cycle | FMG(V) | FMG(W) |
| 2 | 3902 | 4754 | 4487 | 5987 |
| 4 | 1851 | 2264 | 2104 | 2695 |
| 8 | 1026 | 1266 | 1177 | 1515 |
| 16 | 523 | 647 | 613 | 799 |
| 32 | 278 | 354 | 352 | 479 |
| 64 | 182 | 236 | 265 | 384 |

Table 3.1: Time comparison of MG schemes for a simple 2D test problem

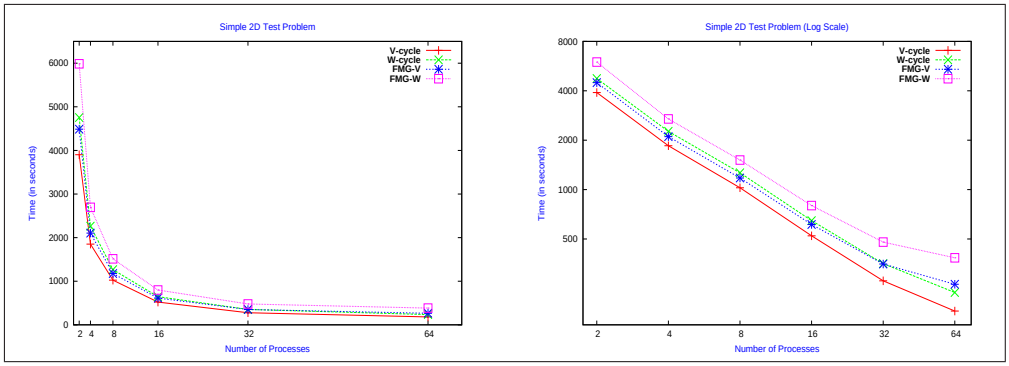


Figure 3.4: Parallel performance and scaling of MG schemes for a simple 2D test problem

The FMG(W) scheme is slightly more expensive in comparison to the other MG schemes but has the same pattern. This difference in performance is

expected and is due to the nature of this MG scheme. Further, plots in a log scale 3.4 (right) shows that, apart from small differences in time, all schemes behave in the same way and scale well up to 32 MPI processes. A slight deterioration in scaling for 64 MPI processes is noticeable however and it is due to the size of sub problem which becomes too small for each MPI process, just two elements per MPI process in any direction, and for this size of sub problem per MPI process, performance and scaling is well within the acceptable margins.

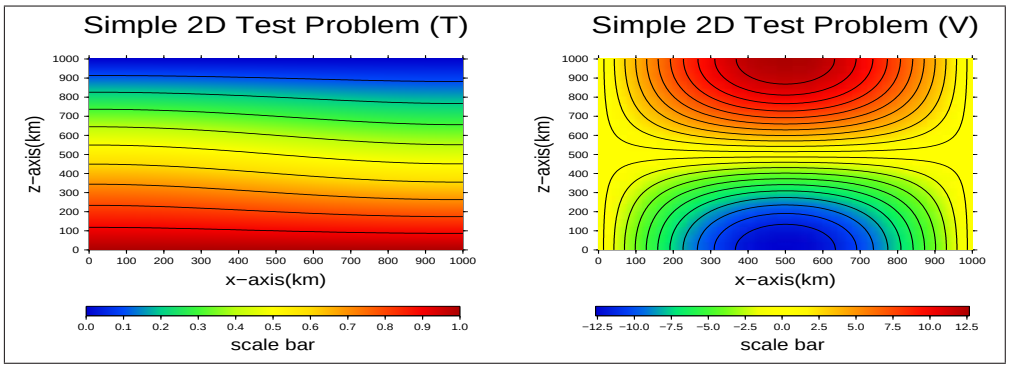


Figure 3.5: Temperature and velocity field charts for a simple 2D test problem

Temperature and velocity fields after the completion of 100 timesteps are shown in figure 3.5. As this is a two dimensional test problem, temperature and velocity fields can only be drawn in the xz-plane as shown.

### 3.5.2 Simple 3D Test Problem

This section consists of results for the simple test problem in 3D. As previously mentioned in section 3.4, this simple test problem in 3D is an extension of the simple test problem in 2D for which the results and discussion can be found in section 3.5.1.
These results are for between 32 and 256 MPI processes. The time taken by each MG scheme to complete 100 timesteps for this test problem in 3D is presented in table 3.2 and is plotted against the number of MPI processes in figure 3.6 (left). This graph shows that V-cycle and FMG(V) schemes perform equally well and slightly better as expected then the corresponding MG schemes based on the W-cycle, that is, W-cycle and FMG(W) schemes.

| Number of Processes | Time (in seconds) | | | |
|---|---|---|---|---|
| | V-cycle | W-cycle | FMG(V) | FMG(W) |
| 32 | 21826 | 24656 | 21935 | 25907 |
| 44 | 13548 | 16354 | 13194 | 16736 |
| 128 | 5851 | 6674 | 5869 | 7039 |
| 256 | 3635 | 4420 | 3586 | 4641 |

Table 3.2: Time comparison of MG schemes for a simple 3D test problem

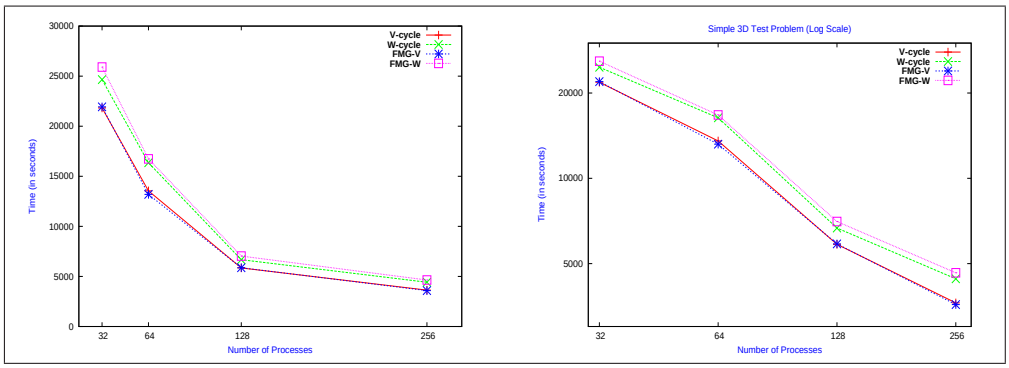


Figure 3.6: Parallel performance and scaling of MG schemes for a simple 3D test problem

The same graph in a log scale 3.6 (right) makes it clear that, apart from negligibly small differences in time, all schemes behave in the same way and scale well for between 32 and 256 MPI processes. For this test problem in 3D, the sub problem size per process remains of reasonable size in each case and shows no deterioration in scaling. In other words, all MG schemes scale well as the number of MPI processes are increased or the sub problem size per MPI process is reduced as long as it does not become too small as seen in case of simple 2D test problem. However, the observed bends are thought to be due to the variation in number of cores per node in use for particular jobs for between 32 and 256 MPI processes.
The temperature and velocity fields after the completion of 100 timesteps are shown in figure 3.7. Due to this being a three dimensional test problem, it is possible to draw these charts for the temperature field in the yz-plane and the xz-plane and for the velocity field in the xz-plane and the xy-plane. These charts are drawn at the surface (zero kilometres), at 400 and 800

kilometres away from the surface. As these are drawn just after completion of 100 timesteps, due to the nature of the test problem, the difference in charts is not easily distinguishable except for the velocity field charts in the xy-plane.

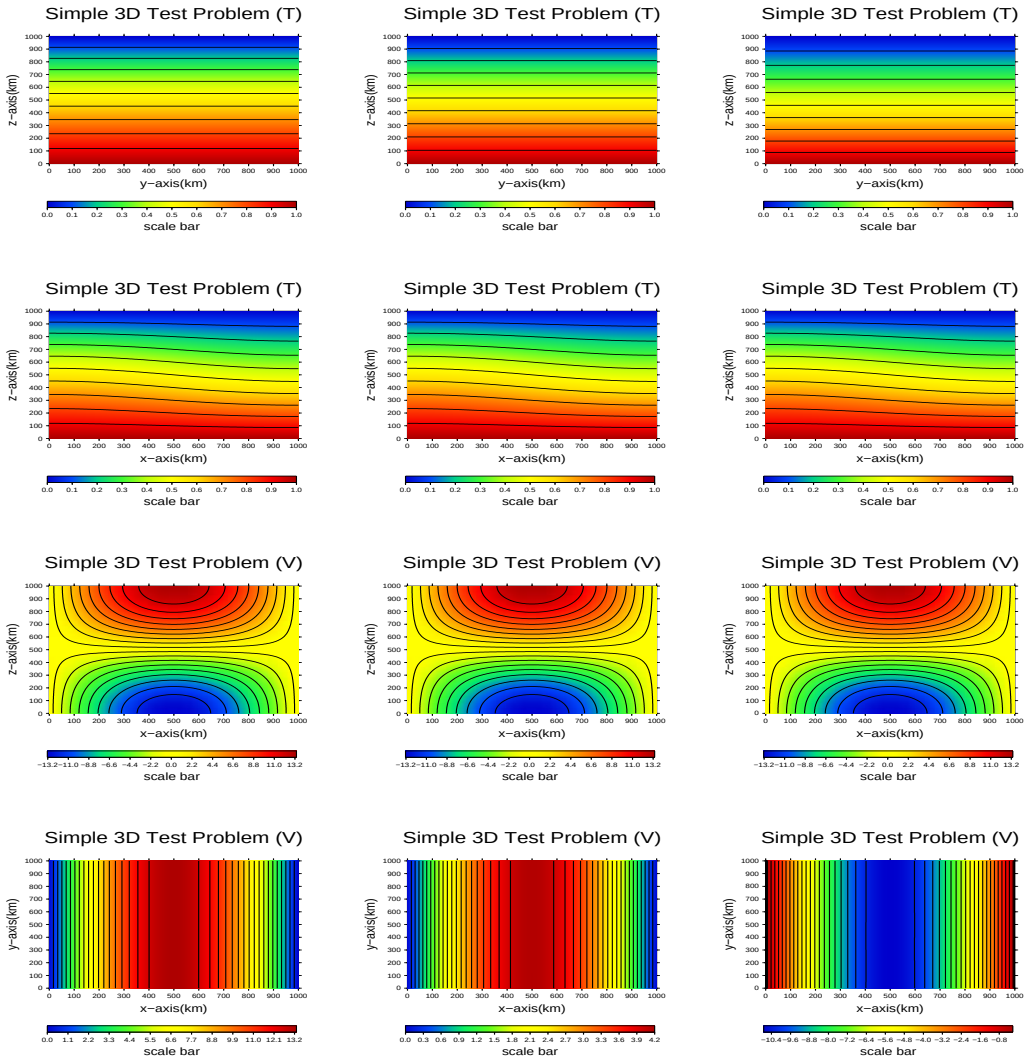


Figure 3.7: Temperature and velocity field charts for a simple 3D test problem

### 3.5.3 Complex 2D Test Problem

This complex 2D test problem is very difficult to solve numerically and helped in identifying the weakness of the V-cycle and W-cycle schemes as both of these schemes failed to converge within 12 hours, the maximum allowed time for any job on HECToR. However, FMG(V) and FMG(W) schemes managed to converge for jobs comprising of between 4 and 64 MPI processes but failed for 2 MPI process jobs.

| Number of Processes | Time (in seconds) | | | |
|---|---|---|---|---|
| | V-cycle | W-cycle | FMG(V) | FMG(W) |
| 2 | - | - | * | * |
| 4 | - | - | 22883 | 23238 |
| 8 | - | - | 14005 | 18396 |
| 16 | - | - | 8934 | 12493 |
| 32 | - | - | 5676 | 8286 |
| 64 | - | - | 5815 | 7600 |

Table 3.3: Time comparison of MG schemes for a complex 2D test problem

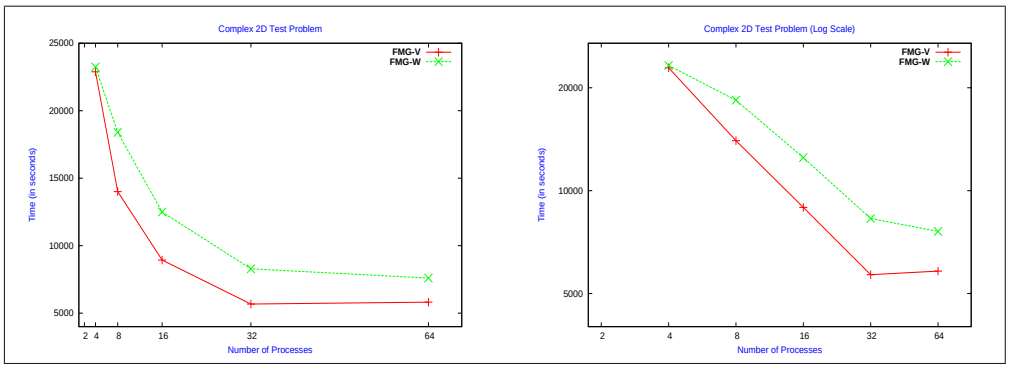


Figure 3.8: Parallel performance and scaling of MG schemes for a complex 2D test problem

The time taken by the FMG(V) and FMG(W) MG schemes to complete 100 timesteps in the case of this complex 2D test problem is presented in

table 3.3 and plotted against the number of MPI processes in figure 3.8 (left) which shows that the FMG(V) scheme is performing better as compared to the FMG(W) scheme for between 8 and 64 MPI processes. The overall performance pattern is almost similar in both cases. In particular, performance for 4 MPI process jobs is nearly the same for both schemes but started deteriorating for 64 MPI process jobs which is acceptable partly due the decreased sub problem size per MPI process. This deterioration is worse for the FMG(V) scheme as it took more time for a 64 MPI process job than a 32 MPI process job and is elaborated in figure 3.8 (right). Apart from this, scaling for up to 32 MPI process jobs is surprisingly very good for these MG schemes.

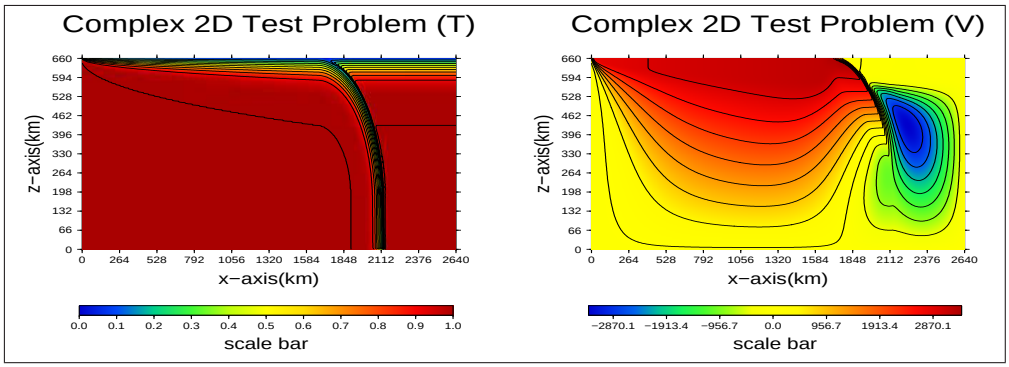


Figure 3.9: Temperature and velocity field charts for a complex 2D test problem

Temperature and velocity fields after the completion of 100 timesteps are shown in figure 3.9. Due to this being a two dimensional test problem, temperature and velocity fields are drawn in the xz-plane only. These charts show a sharp change in phase which might be the reason for the failure of the V-cycle and W-cycle schemes to converge.

### 3.5.4 Complex 3D Test Problem

This complex 3D test problem is difficult to solve in comparison to the simple 3D test problem but is relatively easier to solve than the previously attempted complex 2D test problem. This test problem also helped in identifying some other aspects of the MG schemes.

| Number of Processes | Time (in seconds) | | | |
|---|---|---|---|---|
| | V-cycle | W-cycle | FMG(V) | FMG(W) |
| 32 | 42960 | 31386 | 26940 | 34940 |
| 64 | * | 21205 | 16305 | 22440 |
| 128 | 13167 | 8147 | 7166 | 9306 |
| 256 | 12031 | 5469 | 4423 | 6150 |

Table 3.4: Time comparison of MG schemes for a complex 3D test problem

Time taken by all MG schemes to complete 100 timesteps is presented in table 3.4. It should be noted that the V-cycle failed to complete 100 timesteps within the permitted maximum time of 12 hours for any job on HECToR and could complete only 88 timesteps for a 64 MPI process job. This run was repeated more than once and each time it could only complete 88 timesteps in contrast to the 32 MPI process job which managed to complete 100 timesteps. This V-cycle scheme behaviour for 64 MPI process job is neither understood nor logical but found consistent for many repeated runs.

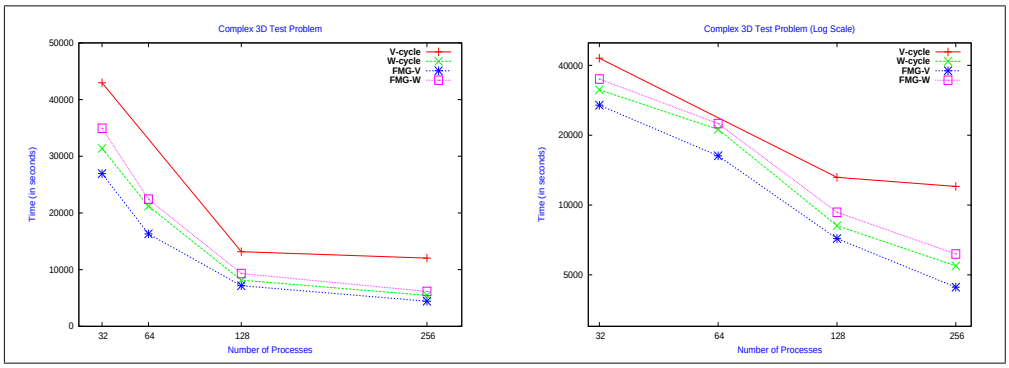


Figure 3.10: Parallel performance and scaling of MG schemes for a complex 3D test problem

These times are plotted against the number of MPI processes in figure 3.10 (left) and show that the FMG(V) scheme is performing better than any other MG scheme for between 32 and 256 MPI processes followed by the W-cycle and FMG(W) schemes leaving the V-cycle scheme at the last. Performance patterns of all schemes are very similar to each other except for the V-cycle

which could not complete 100 timesteps for the 64 MPI process job and started deteriorating for the 256 MPI process job. The other three schemes scale well in contrast to the V-cycle scheme as shown in figure 3.10 (right). This scaling pattern is very similar to that of MG schemes for the simple 3D test problem.

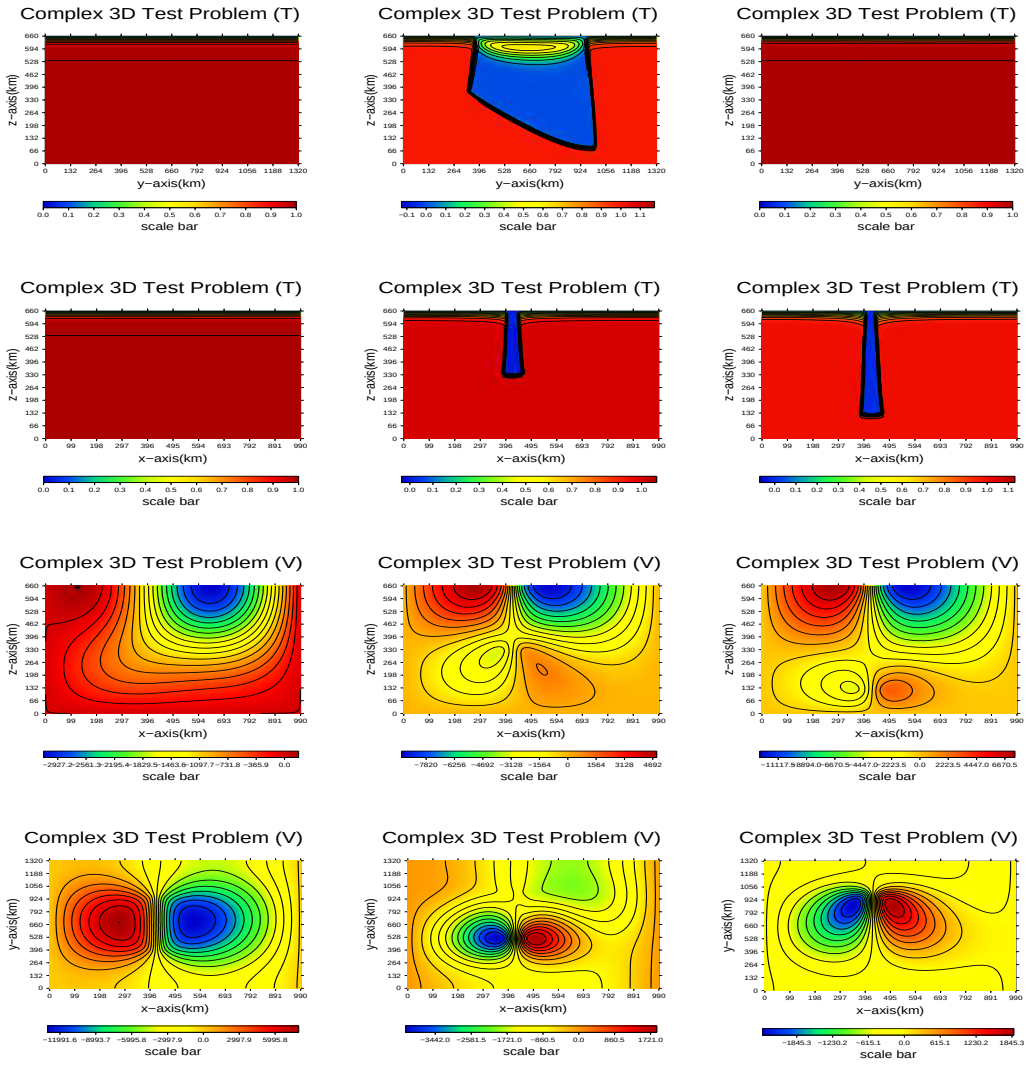


Figure 3.11: Temperature and velocity field charts for a complex 3D test problem

The temperature and velocity fields after the completion of 100 timesteps are shown in figure 3.11 in the yz-plane and xz-plane for the temperature field and in the xz-plane and xy-plane for the velocity field. These charts are drawn at the surface (zero kilometres), at 400 and 600 kilometres away from the surface and show changes to the temperature and velocity fields there.

### 3.5.5 Summary

Analysis of these results for the four representative test problems in the previous sections leads to a few observations summarised as:

- As a result of this dCSE work, CITCOM performs over 31% faster for the V-cycle multigrid scheme, over 38% faster for the W-cycle multigrid scheme in comparison to the corresponding FMG(V) and FMG(W) schemes respectively for the simple 2D test problem and over 12% faster for the W-cycle multigrid scheme in comparison to the corresponding FMG(W) scheme for the complex 3D test problem. These figures are based on best performance cases.

- V-cycle scheme is the fastest MG scheme for relatively simple and easy to converge problems.

- FMG schemes perform well in contrast to the corresponding V-cycle and W-cycle schemes for complex problems. Additionally, later mentioned MG schemes might fail to converge in such cases.

- V-cycle based MG schemes generally perform better than the corresponding W-cycle based MG schemes

- Scaling of any MG scheme that manages to converge is generally excellent. This is particularly true if the sub problem size per MPI process is not reduced extremely to the minimum possible size, that is, just two elements per MPI process in any direction.

- V-cycle scheme offers optimal choice for relatively simple problems and FMG(V) offers optimal choice for relatively complex problems.

- Scaling might be slightly affected by the use of one or two cores per node in comparison to the use of all four cores per node.

## 3.6 Post Processing Tools

This section consists of three types of data processing tools or utilities used in this dCSE project as described below.

### 3.6.1 CITCOM Tools

A brief description of CITCOM post processing tools followed by their usage is given below.

#### 3.6.1.1 combcoord

This program/tool combines the local coordinates information from each partition to build the global coordinates of all nodal points at the finest level of grid.
Syntax of combcoord:

***combcoord*** *<inputfile>*

#### 3.6.1.2 combany

This program/tool combines the first column from files *<basename>.temp.<procID>.<timestep>* corresponding to each partition and builds temperature field corresponding to global node points. In these files, first column contains temperature field related data.
Syntax of combany:

***combany*** *<inputfile> temp <timestep>*

The argument *temp* corresponds to the part of the file name containing data, <timestep> is the output timestep appended at the end of each output data file name.

#### 3.6.1.3 combvel

This program/tool extracts the velocity field from files *<basename>.temp.<procID>.<timestep>* corresponding to each partition and builds a global velocity field that maps to the global set of nodal points. In these files, columns 2, 3 (in 2D) and 4 (in 3D) represents the x, z, (in 2D) and y (in 3D) components of the velocity field.
Syntax of combvel:

***combvel*** *<inputfile> <timestep>*

The tools ***combcoord***, ***combany*** and ***combvel*** use the same order to store values of coordinates, temperature and velocity fields so that the global nodal points correspond to these values correctly. The command-line parameters in all these cases have usual meanings as described:

1. *inputfile* obviously stands for the input file used to run CITCOM job.
2. *temp* is part of the data file name containing the information about a specific field at node points, e.g. `temp` stands for temperature field.
3. *timestep* is the interval at which CITCOM writes data to output files.

#### 3.6.1.4 reggrid

This program/tool reads in data processed with *combcoord* and one of the other two programs/tools, namely *combany* or *combvel*, described above and generates output data with all necessary information about temperature or velocity for example to be used for drawing charts as shown in section 3.5
Syntax of reggrid:

***reggrid*** *<inputfile> <field> <snapshot> <orientation>*
*<coordinate> <discretization> <discretization>*

#### 3.6.1.5 regvector

This program/tool does the same job as *reggrid* except that it adds extra directional arrows describing the direction of flow for temperature or velocity for example.
Syntax of regvector:

***regvector*** *<inputfile> <field> <snapshot> <orientation>*
*<coordinate> <discretization> <vectorscaling>*

In case of ***reggrid*** and ***regvector***, most of the command-line parameters are similar and can be described as:

1. *inputfile* stands for the input file used to run a CITCOM job.
2. *field* is part of the data file name containing the information about a specific field at node points, e.g. `temp` for temperature field.

3. *snapshot* stands for timestep.

4. *orientation* is in `x-`, `z-`, or `y-` coordinate (in 2D or 3D as appropriate) which is kept fixed, so e.g. `y` means a plot in `xz-plane`.

5. *coordinate* is the value of the fixed coordinate, say `y 0` or `z 100`.

   It should be noted that in 2D the only possible option for *orientation* and *coordinate* is `y 0`, that is, a plot in xz-plane only.

6. *discretization*: ***reggrid*** allows for separate discretizations in the two directions of the plane, while ***regvector*** uses the same for both; the value is dimensional and in kilometres.

7. *vectorscaling* is a non-dimensional scaling of the size of vector: the smaller the value, the smaller the vector sizes. A typical value is `0.001`.

### 3.6.2 Gnuplot

Gnuplot [13] is a copyrighted but freely distributed software, portable and command-line driven, used for interactive data and function plotting on UNIX, MS Windows, DOS, and many other platforms. Gnuplot supports many types of plots in 2D and 3D. It can draw using lines, points, boxes, contours, vector fields, surfaces, and various associated text. It also supports various specialised plot types. Gnuplot supports many different types of output: interactive screen terminals, direct output to pen plotters or modern printers, and output to many file formats, for example, eps, fig, jpeg, LaTeX, metafont, pbm, pdf, png, postscript, svg, etc. All graphs presented in this report in section 3.5 are drawn using Gnuplot.

### 3.6.3 The Generic Mapping Tools (GMT)

The Generic Mapping Tools or GMT [14] is an open source collection of more than 60 programs for manipulating geographic and Cartesian data sets including filtering, trend fitting, gridding, projecting, etc. and producing (encapsulated) postscript file illustrations ranging from simple `xy` plots via contour maps to artificially illuminated surfaces and `3D` perspective views. GMT supports nearly 30 map projections and transformations and comes with a variety of support data. Charts given in section 3.5 are generated using a few of the GMT components which are described in table 3.5.

| Program | Description |
|---|---|
| grdcontour | Contouring of 2-D gridded data sets |
| grdimage | Produce images from 2-D gridded data sets |
| makecpt | Make color palette tables |
| psbasemap | Create a basemap plot |
| psscale | Plot gray scale or colour scale on maps |
| xyz2grd | Convert an equidistant table xyz file to a 2-D grid file |

Table 3.5: GMT programs and their description.

## 3.7 Next Phase

Next phase (phase 3) of this dCSE project consists of two tasks:

1. Mesh refinement near high viscosity gradients.
2. Improved prolongation and restriction

These tasks are explained briefly along with some questions on implementation feasibility within CITCOM code structure.

### 3.7.1 Mesh Refinement

Mesh refinement in MG methods is clearly distinct from the common procedure of smoothly decreasing the element size towards the location where velocity or viscosity gradients are large. Such a method is very difficult to combine with a structured mesh that requires a nested set of MG meshes. However, MG naturally offers a different local mesh refinement method. By locally adding one or more finer layers, the resolution can locally be improved by a factor of 2 or powers of 2. One such possible scenario of uniform local refinement is depicted in figure 3.12(left) where only the boundary elements are refined.
The other scenario of non-uniform local refinement which is more complicated and difficult to implement is depicted in figure 3.12(right). Since the grid elements at the finer MG levels do not exist globally, these levels require some special treatment, and introduce some complexity to the solution methods as well. In order to handle this complexity there might be a need to introduce

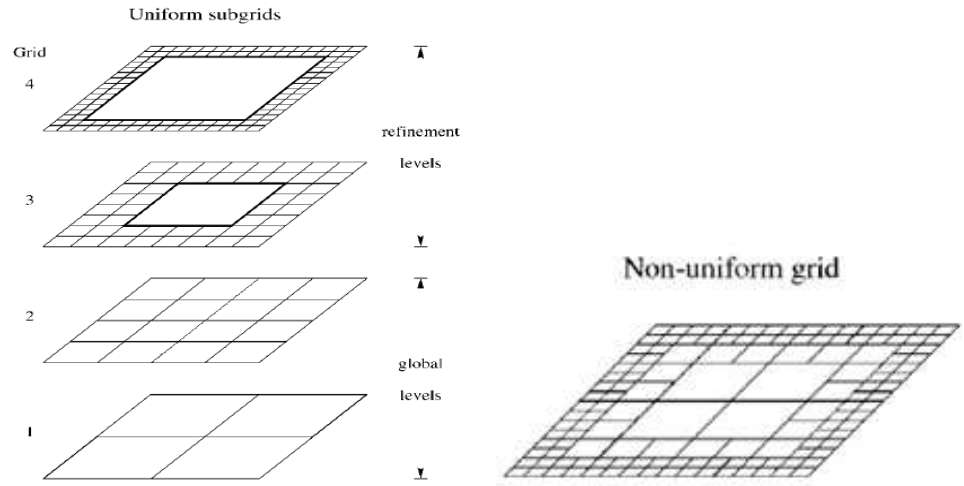


Figure 3.12: Examples of uniform (left) and non-uniform (right) refinement

an extra layer of "ghost nodal points" on the interior boundaries and/or extra bookkeeping of the information at the finer grid levels which do not exist everywhere. This would increase memory requirements significantly and lead to load imbalance in most cases.
Due to importance of the local mesh refinement and the envisaged improvement it can make to the rate of convergence, the implementation of this scenario is the top priority in the next phase.

### 3.7.2 Improvements to Prolongation and Restriction

Restriction and prolongation operations may help to improve the rate of convergence by adopting different averaging schemes [1] as suggested in the proposal. However, studying the current implementations for restriction (based on weighted averaging) and prolongation (based on weighted interpolation), previous experience of the PI and outcome of the discussion with the PI on this topic suggests that it is highly unlikely to gain any significant advantage by implementing suggested averaging schemes over the existing implementation.
Therefore it is agreed that the work in phase 3 would concentrate on the implementation of the local mesh refinement.

# Chapter 4

# Refinement

This chapter covers two topics of interest as suggested in the project proposal:

1. Prolongation and Restriction
   These operations may help improve the rate of convergence.

2. Local Mesh Refinement
   For example a better convergence rate may be achieved by refining part of the mesh in regions with high velocity gradients.

Each of these topics is discussed below.

## 4.1 Prolongation and Restriction

For the last phase (phase 3) of the project, two work packages are defined to implement: (i) local mesh refinement and; (ii) improvements to prolongation and restriction operations by way of implementing arithmetic, geometric and harmonic averaging and matrix-dependent transfer. The topic of prolongation and restriction required investigation before it could be implemented in the existing scenario so that it is of benefit to CITCOM. Prolongation and restriction operations help update the nodal values for the nodes at upper and lower levels in the multigrid context with nodal values from lower and upper levels respectively. An appropriate implementation of these operations, by adopting different averaging schemes [1] is expected to help improve the rate of convergence.
In this context, the study of current implementations for restriction (based on weighted averaging) and prolongation (based on weighted interpolation)

along with the previous experience and discussion with PI on this topic helped in reaching the conclusion that it is unlikely to gain any significant advantage by implementing suggested averaging schemes over the existing implementation. Therefore, it was agreed (with PI and NAG) that any implementation of prolongation and restriction operations as suggested in the proposal may not be of any advantage. Following on from this decision, the remaining time of nearly four weeks for this current phase was allocated to local mesh refinement. The initial part of it had been spent studying the current implementation, and on exploring ways to help the implementation as suggested in the original proposal.

## 4.2 Local Mesh Refinement

Following on from the basic concepts introduced in section 3.7.1, the intention here is to extend the concept in the context of the adopted scheme for implementation within CITCOM as detailed below.

### 4.2.1 Refinement Strategy

One way to express the challenge of local mesh refinement may be: to increase the spatial resolution of some parts of the domain or sub-domain which are dynamically selected following some specific criteria. Of course the choice of these parts is not obvious, nevertheless, for the general description of the method, we have assumed that a refinement criteria is available and the refinement area is known before hand. In this particular case, this refinement area is a certain fraction of the z-dimension covering at least two elements or an even number of elements if there are more than two elements in the base level (level 0) mesh. To avoid the load imbalance situation and to capture most of the sharp gradients e.g. for velocity, which occur near the earth's surface where the lithosphere is located, all elements in x- and y-dimension are refined. This yields the lower part of the mesh as unrefined or coarse and the upper part of the mesh (near and at the earth's surface) as refined at base level. Once this local refinement of the mesh near the top of domain is achieved at base level, which now consists of elements of two different size: (i) coarse elements and (ii) refined elements achieved by bisecting a coarse element in to two sub elements in each dimension yielding four refined elements in two dimension and eight refined elements in three dimension, all

of the mesh is refined for the higher level meshes in the nested mesh hierarchy. For more information on this point a revisit to the sub section 2.2.1 would be of help to understand the way the Cartesian coordinate system is used within CITCOM.
Due to the fact that no specific local mesh refinement strategy for implementation is identified in the proposal. We have discussed the pointers to the ways it can be achieved to a certain depth, detail and a number of possible schemes for local mesh refinement in the context of the regular structured mesh elements used in CITCOM. To conclude this we have decided to implement the *one-level-difference* refinement rule as it fulfils the requirement of only one level of refinement at the top of the domain without triggering the load imbalance problem.

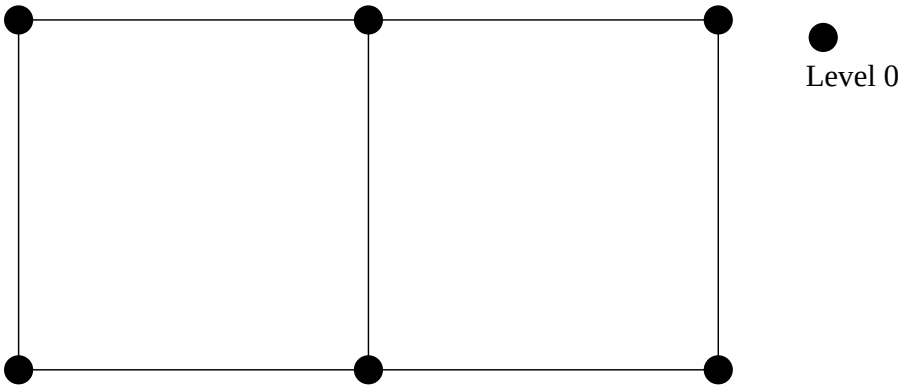


Figure 4.1: A two element mesh in 2D at level 0

Here we describe the *one-level-difference* refinement rule based on local mesh refinement in more detail. In order to illustrate the local mesh refinement we have used, we shall show refinement of squares/rectangles in two dimension and cubes/cuboids in three dimensions. Figure 4.1 consists of a two element mesh in two dimensions and shall be used to explain the local mesh refinement where all mesh elements and nodes are at level 0.
Let us assume that we wish to refine this mesh up to two levels with refinement at and near the right-bottom corner of the mesh. This implies that any element near the right-bottom corner of the mesh at level 0 is a candidate for refinement and the resulting new elements from the refinement of that element form the mesh at level 1. Subsequently, any element at level

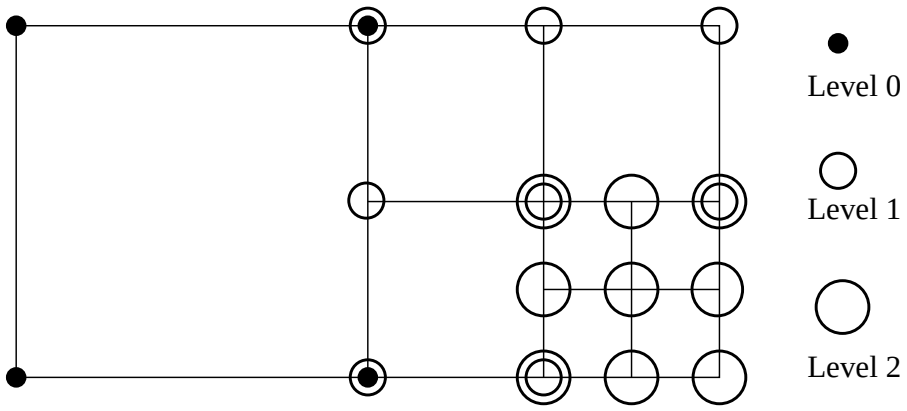


Figure 4.2: A local refinement without obeying *one level difference* rule

1 at or near the right-bottom corner is refined again and the new elements are constructed to make up the mesh at level 2. This scenario is depicted in figure 4.2 for elements in two dimensions where refined or sub elements are obtained by bisecting a coarse level element in each dimension, in this example along the x- and y-dimensions, to generate four new elements each time an element is refined. After two levels of refinement of the elements near the right-bottom corner, we get a total of eight elements; (i) one element at level 0, (ii) three elements at level 1, and (iii) four elements at level 2. The corresponding nodes at level 0 are marked with a black dot, at level 1 with a small circle and at level 2 with a larger circle. Any node shared by two elements which are at different levels are marked by a black dot and a small circle or a small and a large circle accordingly. However we note that this refinement strategy does not obey the *one-level-difference* refinement rule.
From the point of view of refinement although it is perfectly acceptable but in the given multigrid context, it is not workable. It makes the mapping between different mesh levels hard and it becomes harder and harder (if not impossible) as the mesh is further refined to higher levels. Therefore, it is not the most suitable refinement scheme for CITCOM.
Another possible scenario is to refine the element at or near the right-bottom corner to the desired refinement level, which is level 2 in this case, as shown in figure 4.3. This creates elements which are at the two extremes in the mesh; (i) the largest elements at coarse level and (ii) elements which are $4^n$ times smaller in two dimensions and $8^n$ times smaller in three dimensions than

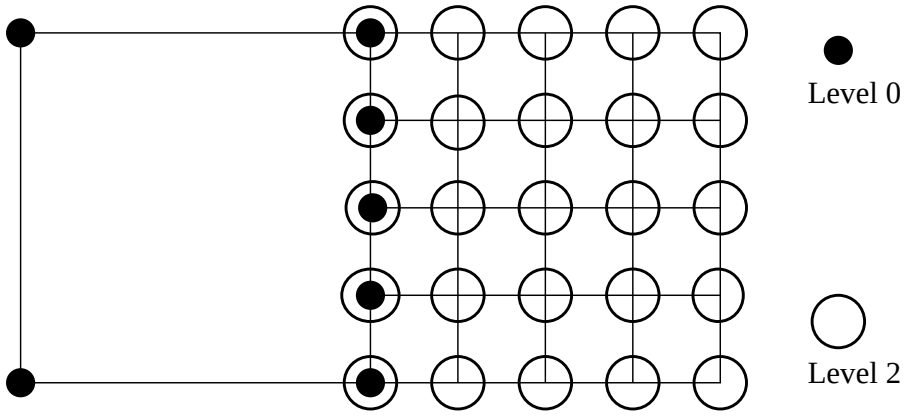


Figure 4.3: A local mesh refinement forbidden by *one level difference* rule

the coarse level elements where $n$ is the highest level of refinement. This refinement scheme is forbidden by the *one-level-difference* refinement rule. Additionally, this creates a larger number of hanging nodes on the edge(s) of coarser elements, this is an undesired situation as handling any hanging nodes not only requires extra work but is hard to deal with too.
Let us again assume that we wish to refine the same two element mesh up to two levels with refinement at or near the right-bottom corner of the mesh. However, this time we obey the *one-level-difference* refinement rule, that is, any two elements in the fully refined mesh (up to level 2 in this case) are at levels which are immediately next to each other in the nested hierarchy of the refined meshes. In other words, no element is at level 0 when this two element mesh is refined to level 2 as shown in figure 4.4. This shows that elements in the mesh after refinement are at the difference of a maximum of one level from any other element in the mesh. Therefore, to refine the elements at or near the right-bottom corner of the mesh to level 2, other elements in the mesh, which would have not been refined otherwise, must be refined at least up to level 1. This is an additional restriction to the *one-level-difference* refinement rule described in [10] and [3]. This ensures that all elements and nodes in the mesh are either at level *1* or level *2* avoiding the complex situation of multilevel nesting of mesh elements which becomes very hard to handle if not impossible in multigrid prolongation and restriction operations.
We note that in order to achieve the mesh refinement shown in figure 4.4,

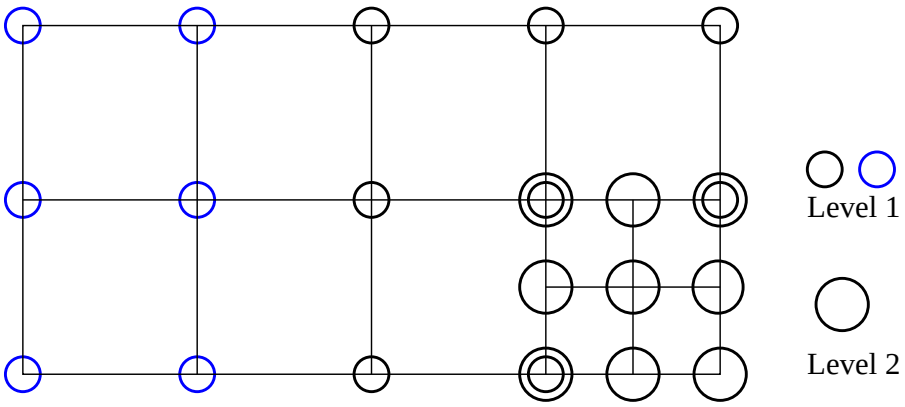


Figure 4.4: A local refinement by *one level difference* rule

all elements at level 0 must be refined when an element at or near the right-bottom corner is refined a second time. Elements whose nodes are marked with blue circles represent those elements which are refined as a consequence of refinement of other elements in the mesh. This can easily be generalised so that all elements at level *n-1* must be refined when any element at level $n$ is refined so that all elements in the resulting mesh are either at level $n$ or *n+1*.
Having described the strategy for local mesh refinement by following the *one-level-difference* refinement rule, we setup some principals and restrictions in the context of the current implementation in CITCOM. It is assumed that:

- Only part of the mesh along the z-axis, in two and three dimension, is refined. Recall that Cartesian coordinates in CITCOM are defined as $(x,z,y)$ in contrast to the customary definition as $(x,y,z)$.
- Part of the mesh which is refined has elements at one level higher than the level of elements which are not refined.
- After local refinement of the mesh at level 0, base level coarse elements and new elements resulting from refinement are considered at level 0. Refinement to the next higher level is carried out throughout the mesh to maintain the similar mesh structure at each level.
- A node on a coarse element edge which is at the interface with refined elements is dealt by the refined or higher level elements only. For the

coarse element, any such node is treated as non existent.

- Each mesh level consists of elements of two sizes; coarser elements and refined elements although both at the same level.
- A node on a coarse element edge which is shared by refined elements (at least two or more) is accounted for the refined elements only. Therefore, the overall mesh is considered as having no hanging nodes.

### 4.2.2 Refinement Setup

Following on from the strategy described in section 4.2.1, here we describe the local mesh refinement implemented in CITCOM, with the help of an example mesh in two dimensions. To start with, we have chosen a basic mesh of a reasonably small size in two dimensions with eight elements in the x-dimension and six elements in the z-dimension. It is distributed into four sub-domains as shown by the dark lines in figure 4.5.

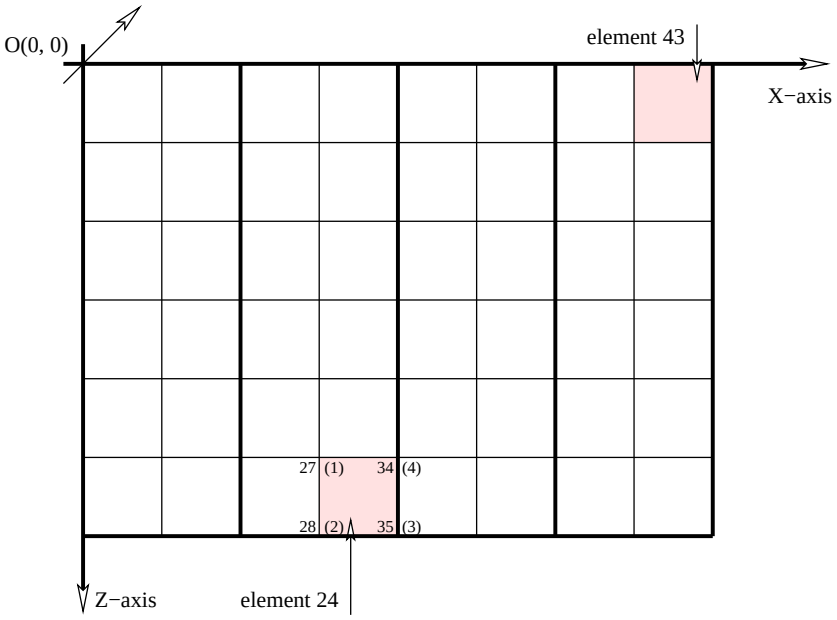


Figure 4.5: A coarse mesh of 48 elements in two dimension

In order to solve each sub-domain problem, we employ one process per sub-domain so that four processes are employed to solve the global problem; four along the x-dimension and one along the z-dimension, that is, each process covers the whole length along z-axis. Note that CITCOM does not allow more than one process in the z-dimension. As a by product, this helps in keeping the load of all processes reasonably balanced and no communication is required in the z-direction. This is also one of the reasons for choosing this *one-level-difference* refinement rule. The connectivity for a representative finite element numbered as 24 is given by node numbers $27, 28, 35$ and $34$ corresponding to the local nodes numbers $1, 2, 3$ and $4$ respectively. This also reveals that element numbering in the mesh starts from 1 near the origin $(0, 0)$ and is incremented in the z-direction followed by the x-direction. Similarly, (global) node numbering follows the same pattern whereas local numbering is anti-clockwise. This is extended to include the y-axis as a third dimension in the case of three dimensions as shown in figure 4.6. For the sake of clarity this is for a relatively smaller mesh size.

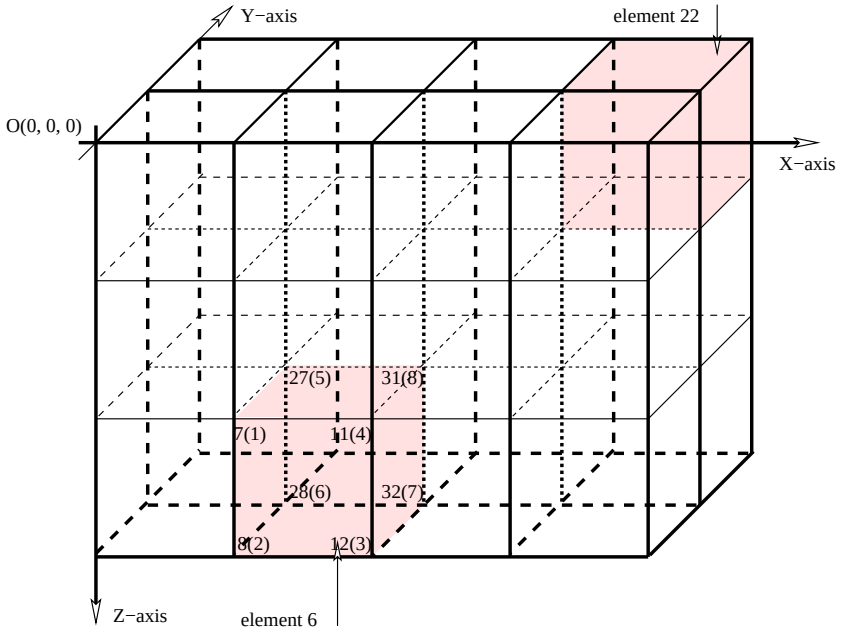


Figure 4.6: A coarse mesh of 24 elements in three dimension

Following the *one-level-difference* refinement rule and assumptions described

in section 4.2.1, the mesh shown in figure 4.5 is refined at the top of the domain covering two elements in the z-direction, all eight elements in the x-direction and all two elements in the y-direction. This local refinement replaces existing elements with the new elements which are half the size and twice the number of the original elements in all dimensions. In other words, each refined element is replaced by four smaller elements of *one-fourth* the size of the original element in two dimensions as shown in figure 4.7 and by eight smaller elements of *one-eighth* the size of original element in three dimensions as shown in 4.8 (for a relatively smaller mesh size for the sake of clarity) respectively.

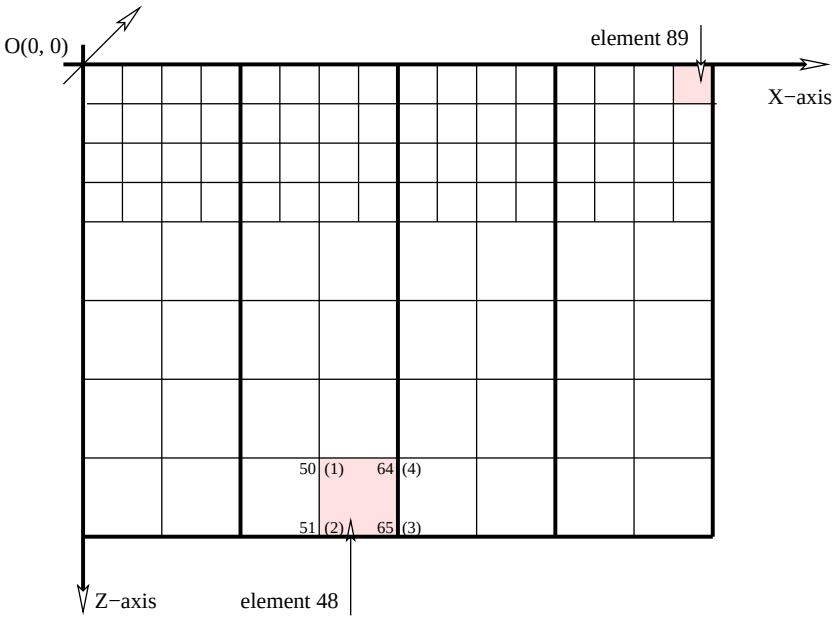


Figure 4.7: Local refinement of the coarse mesh at top of the domain in two dimension

However, it is important to note that these new elements are at the same level as the parent elements, which is the base level or level 0. After the setup of this base level mesh following the local mesh refinement, every next level in the nested mesh hierarchy is obtained by refining all elements in the mesh so that at any given mesh level, elements at the top of the domain are always smaller than the other elements in the mesh at that level but all elements are at the same level.

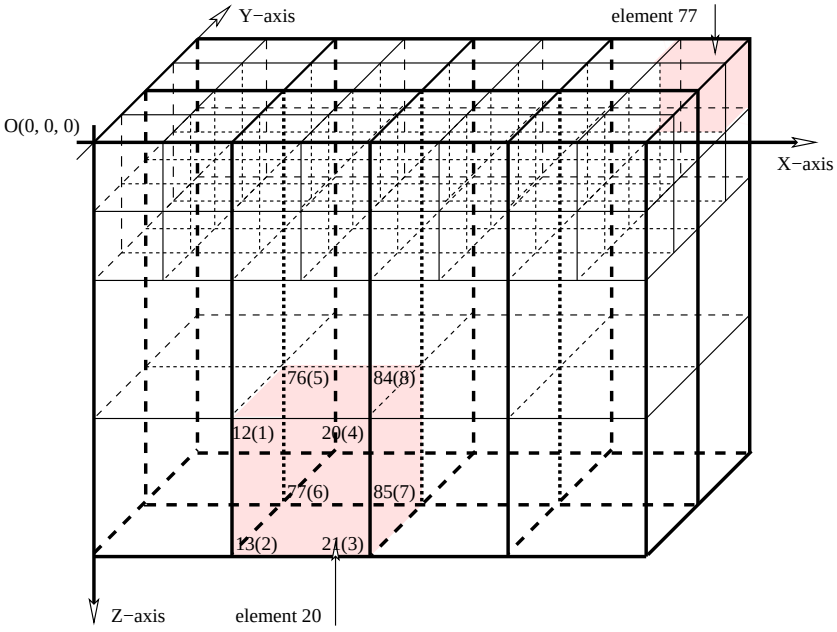


Figure 4.8: Local refinement of the coarse mesh at top of the domain in three dimension (for a relatively smaller mesh)

In order to specify the refinement area at the base level mesh, a bounding box is defined in the CITCOM input file and consists of two or three pairs of values, namely, $(xmin, xmax)$, $(zmin, zmax)$ and $(ymin, ymax)$ in two dimensions or three dimensions respectively. As these values are defined in the input file, the mesh refinement area is controlled at run time which offers the extra flexibility of changing the refinement area for experimentation without recompiling the whole source code.
The implementation of local mesh refinement in CITCOM required a major rewrite of a large number of functions, particularly related to the setup of C structures defined in CITCOM which provides the backbone to the functionality and construction of elements, sub elements, nodes, surface nodes, neighbouring nodes, node maps, coordinates, domain decomposition, neighbouring sub domains, communication routes, boundary conditions, setup for prolongation and restriction operations, etc. A brief description of a couple of these structures, their setup and how to access these structure members, modifications to the code, etc. are described in the following section.

### 4.2.3 Refinement Implementation

In this section, we try to reflect on the C structures defined in CITCOM source code by using snapshots of just two out of a total of 62 structures. These are in addition to a large number of macros and *look-up tables* which are used to construct and fill-up these and other structures and arrays used for a variety of computational tasks. For example, populating of `IEN struct` with global node numbers for a given mesh level and element number with the help of a look-up table for the local node numbers of an element and then using it to populate the `NEI struct` with the number of elements surrounding any given node and mesh level and so on. One of these structures is `All_variables`, the top level structure is given below. It shows that a large number of other structures are nested in it along with other variables. The size of these nested structures vary ranging from having a few to a large number of *member variables*.

**struct All_variables**

```
struct All_variables {                                                  1
   struct TRACERS            tracers;                                   2
   struct HAVE               Have;                                      3
   struct BAVE               Bulkave;                                   4
   struct TOTAL              Total;                                     5
   struct MESH_DATA          mesh;                                      6
   struct MESH_DATA          lmesh;                                     7
   struct CONTROL            control;                                   8
   struct OUTPUT             output;                                    9
   struct MONITOR            monitor;                                   10
   struct DATA               data;                                      11
   struct SLICE              slice;                                     12
   struct Segment            segment;                                   13
   struct Slabs              slabs;                                     14
   struct Crust              crust;                                     15
   struct Parallel           parallel;                                  16
   struct INFO               info;                                      17
   struct Shape_function     N;                                         18
   struct Shape_function_dx  Nx;                                        19
   struct Shape_function1    M;                                         20
   struct Shape_function1_dx Mx;                                        21
   struct Shape_function1    L;                                         22
   struct Shape_function1_dx Lx;                                        23
   struct NEI                NEI[MAX_LEVELS];                           24
   struct COORD              *eco;                                      25
   struct IEN                *ien;                                      26
   struct SIEN               *sien;                                     27
   struct ID                 *id;                                       28
   struct LM                 *lmd;                                      29
   struct LM                 *lm;                                       30
   struct Shape_function_dx  *gNX;                                      31
   struct Shape_function_dA  *gDA;                                      32
   struct COORD              *ECO[MAX_LEVELS];                          33
   struct IEN                *IEN[MAX_LEVELS];                          34
   struct ID                 *ID[MAX_LEVELS];                           35
   struct SUBEL              *EL[MAX_LEVELS];                           36
   struct EG                 *elt_del[MAX_LEVELS];                      37
   struct EK                 *elt_k[MAX_LEVELS];                        38
   struct FNODE              *TWW[MAX_LEVELS];                          39
   struct LM                 *LMD[MAX_LEVELS];                          40
   struct Shape_function_dx  *GNX[MAX_LEVELS];                          41
   struct Shape_function_dA  *GDA[MAX_LEVELS];                          42
                                                                        43
   FILE                      *fp;                                       44
```

```
    FILE                    *filed[20];                                          45
                                                                                 46
    higher_precision        *Eqn_k1[MAX_LEVELS];                                 47
    higher_precision        *Eqn_k2[MAX_LEVELS];                                 48
    higher_precision        *Eqn_k3[MAX_LEVELS];                                 49
                                                                                 50
    int                     *surf_node;                                          51
    int                     *surf_element;                                       52
    int                     *mat;                                                53
    int                     *Node_map[MAX_LEVELS];                               54
    int                     *Node_eqn[MAX_LEVELS];                               55
    int                     *Node_k_id[MAX_LEVELS];                              56
                                                                                 57
    unsigned int            *node;                                               58
    unsigned int            *eqn;                                                59
    unsigned int            *NODE[MAX_LEVELS];                                   60
    unsigned int            *EQN[MAX_LEVELS];                                    61
                                                                                 62
    float                   *stress;                                             63
    float                   *Psi;                                                64
    float                   *NP;                                                 65
    float                   *Mass;                                               66
    float                   *tw;                                                 67
    float                   *Vi;                                                 68
    float                   *EVi;                                                69
    float                   *Vielem;                                             70
    float                   *T;                                                  71
    float                   *buoyancy;                                           72
    float                   *Tdot;                                               73
    float                   *EEDOTSQR;                                           74
    float                   *dummy;                                              75
    float                   *edummy;                                             76
    float                   *C;                                                  77
    float                   *V[4];                                               78
    float                   *V1[4];                                              79
    float                   *Vest[4];                                            80
    float                   *X[4];                                               81
    float                   *XL[4];                                              82
    float                   *VB[4];                                              83
    float                   *TB[4];                                              84
    float                   *edot[4][4];                                         85
    float                   *MASS[MAX_LEVELS];                                   86
    float                   *VI[MAX_LEVELS];                                     87
    float                   *EVI[MAX_LEVELS];                                    88
    float                   *TW[MAX_LEVELS];                                     89
    float                   *XX[MAX_LEVELS][4];                                  90
                                                                                 91
    double                  *P;                                                  92
    double                  *F;                                                  93
    double                  *H;                                                  94
    double                  *S;                                                  95
    double                  *U;                                                  96
    double                  *temp;                                               97
    double                  *global_F;                                           98
    double                  *dudx[4][4];                                         99
    double                  *tau[4][4];                                          100
    double                  *dtau[4][4];                                         101
    double                  *spin[4][4];                                         102
    double                  *BI[MAX_LEVELS];                                     103
    double                  *BPI[MAX_LEVELS];                                    104
    double                  **global_K;                                          105
    double                  **factor_K;                                          106
};                                                                               107
```

struct All_variables

In this `All_variables` structure, amongst others, `struct Parallel` is a nested structure at line 16 and consists of a few variables and other nested structures as its members as shown below.

struct Parallel

```
struct Parallel {
   char                    machinename[160];
   int                     me;
   int                     nproc;
   int                     nprocx;
   int                     nprocz;
   int                     nprocy;
   int                     nprocxy;
   int                     nproczy;
   int                     nprocxz;
   int                     automa;
   int                     idb;
   int                     log;
   int                     num_b;
   int                     me_loc[4];
   int                     TNUM_PASS[MAX_LEVELS];
   int                     START_ELE[MAX_LEVELS];
   int                     START_NODE[MAX_LEVELS];
   int                     *IDD[MAX_LEVELS];
   int                     *ELE_ORDER[MAX_LEVELS];
   int                     *NODE_ORDER[MAX_LEVELS];

   struct                  BOUND  NUM_NNO[MAX_LEVELS];
   struct                  BOUND  NUM_PASS[MAX_LEVELS];
   struct                  BOUND  NUM_ELE[MAX_LEVELS];
   struct                  PASS   NUM_NEQ[MAX_LEVELS];
   struct                  PASS   NUM_NODE[MAX_LEVELS];
   struct                  PASS   PROCESSOR[MAX_LEVELS];
   struct                  BOUND  *NODE[MAX_LEVELS];
   struct                  BOUND  *IDPASS[MAX_LEVELS];
   struct                  PASS   *EXCHANGE_ID[MAX_LEVELS];
   struct                  PASS   *EXCHANGE_NODE[MAX_LEVELS];
};
```

struct Parallel

A code snapshot to find a (global) node number anywhere in the mesh prior to implementing local mesh refinement is given below. A very simple procedure is followed: loop over the number of nodes `noy` in the y-direction followed by a nested loop over the number of nodes `nox` in the x-direction and finally a third nested loop over the number of nodes `noz` in the z-direction which is not unusual in any such situation. At this point, (global) `node` number is determined using a simple formula given at line 4 of the code snapshot. This becomes even more simple in the two dimension case when looping over nodes in the y-direction has no affect and the first term on the right hand side of the equation at line 4 in the code snapshot vanishes. Looking at figure 4.5 in two dimension and figure 4.6 in three dimension, it is not hard to find a (global) `node` number for any given value of `ix, jz` and `ky`, the node numbers in the x-, z- and y-directions respectively.

Finding global node without local mesh refinement

```
for (ky=1; ky <= noy; ky++) {
  for (ix=1; ix <= nox; ix++) {
    for (jz=1; jz <= noz; jz++) {
      node = (ky-1)*nox*noz + (ix-1)*noz + jz;
    }/* end of for jz=1 */
  }/* end of for ix=1 */
}/* end of for ky=1 */
```

Finding global node without local mesh refinement

Here the same is repeated for the local mesh refinement case. Loops over `noy, nox` and `noz`, nodes in the y-, x- and z-directions respectively are the same. However, we have to have additional checks in place to make sure that we are accounting for refined and coarser parts of the mesh accordingly. For this purpose we have defined extra variables (also members of `struct INFO`):

- znmax - maximum number of nodes in z-direction.
- znbnd - number of nodes in refined part of mesh (band) in z-direction.
- zxnd1 - number of nodes in zx-plan covering coarse and refined part of mesh.
- zxnd2 - number of nodes in zx-plan covering only refined part of mesh.

A few other local variables such as `ynodes, xnodes, xnalt` and `znalt` are also used. Looking at this code snapshot, figure 4.7 in two dimension and figure 4.8 in three dimension, it is clear that for these loops over `noy, nox` and `noz`, the number of nodes in the y-, x- and z-directions respectively, results in accumulating node numbers which are used to determine the (global) `node` number. The incremental contribution within each loop depends on these loop counters being odd, even, or some other combination of these.

Finding global node with local mesh refinement

```
int znmax = E->info.znmax;
int znbnd = E->info.znbnd;
int zxnd1 = E->info.zxnd1;
int zxnd2 = E->info.zxnd2;

ynodes = 0;
for (ky=1; ky <= noy; ky++) {
  ( 1==(ky%2) ) ? (xnalt=zxnd1) : (xnalt=zxnd2);
  xnodes = 0;
    for (ix=1; ix <= nox; ix++) {
      if ( ky%2 ) {
        ( 1==(ix%2) ) ? (znalt=znmax) : (znalt=znbnd);
      }
      else {
        znalt = znbnd;
      }
      for (jz=1; jz <= znalt; jz++) {
        node = ynodes + xnodes + jz;
      }/* end of for jz=1 */
      xnodes += znalt;
    }/* end of for ix=1 */
  ynodes += xnalt;
}/* end of for ky=1 */
```

Finding global node with local mesh refinement

In the previous two paragraphs, with the help of code snapshots, we have compared how to determine a (global) node number before and after the

local mesh refinement. Here we would compare the differences in setting up `struct EL` which determines sub elements for any given element and mesh level in the prior and post local mesh refinement context. In the former case, we have three nested loops (in three dimensions) over the number of elements `ely, elx` and `elz` in the y-, x- and z-directions respectively. The corresponding code snapshot is given below. Note that each element (being a parent) has two sub elements in each dimension: a total of four sub elements in two dimensions and eight elements in three dimensions. With the help of simple formulae, as before for the (global) node numbers, the number of each `element` and that of its first `sub element` is determined (see line number 11 and 12 in code snapshot). This and remaining sub elements are than fed to `struct EL` utilising a pre-defined *look-up table* and loop over sub elements. This procedure is repeated over all mesh levels, except the highest level, for all elements so that on completion we are in a position to access any sub element for any element at any mesh level except the top level.

Setup of struct EL prior to local mesh refinement

```
for (lev=E->mesh.levmax-1; lev >= E->mesh.levmin; lev--) {
   elx = E->lmesh.ELX[lev];
   elz = E->lmesh.ELZ[lev];
   ely = E->lmesh.ELY[lev];
   elxu = 2*elx;
   elzu = 2*elz;

   for (ye=1; ye <= ely; ye++) {
      for (xe=1; xe <= elx; xe++) {
         for (ze=1; ze <= elz; ze++) {
            element = (ye-1)*elz*elx + (xe-1)*elz + ze;
            sub_element = 2*(ye-1)*elzu*elxu + 2*(xe-1)*elzu + (2*ze - 1);

            for (el=1; el <= ENODES[E->mesh.nsd]; el++) {
               E->EL[lev][element].sub[el] = ( sub_element +
                                               offset[el].vector[z] +
                                               offset[el].vector[x]*elzu +
                                               offset[el].vector[y]*elzu*elxu );
            }/* end of for el=1 */
         } /* end of for ye=1 */
      } /* end of for ze=1 */
   } /* end of for xe=1 */
}/* end of lev= ... */
```

Setup of struct EL prior to local mesh refinement

The procedure to fill-up `struct EL` for sub elements in the post local mesh refinement scenario is not as simple as without local mesh refinement. The relevant code snapshot together with a brief dissection is given below.
In this case of local mesh refinement, we have, as previously, three nested loops (in three dimensions) over a number of elements `ely, elx` and `elz` in the y-, x- and z-directions respectively. We then determine the number of each `element` using a simple formula shown on line 24 of the code snapshot.

In two dimensions, the loop over `ely` becomes ineffective. Note that the mesh elements in this case are of two different sizes: the refined part of the mesh consists of elements of one-fourth in size of elements in the coarse part of the mesh in two dimension and of one-eighth in size in three dimensions. The `element` number is determined for the mesh which is not yet refined. At this point, we need to differentiate between elements if they belong to the coarse or the refined part of the mesh and treat accordingly. This is determined with the help of a simple utility function. If an element belongs to refined part of the mesh, its sub elements or, more appropriately, corresponding refined elements which form the actual refined mesh are determined. These refined elements are at the same level (i.e. the current one) as those which belong to the coarse part of the mesh.
Before proceeding further, let us define some variables which are used in filling-up the `struct EL`.

- (xemin, zemin, yemin) - number of elements at current mesh level without refinement in x-, z- and y-directions respectively.
- (xebnd, zebnd) - number of elements at current mesh level in refined part of mesh (band) in x- and z-directions respectively.
- (xeminN, zeminN, yeminN) - number of elements at next upper level without refinement in x-, z- and y-directions respectively.
- (xebndN, zebndN) - number of elements at next upper level in refined part of mesh (band) in x- and z-directions respectively.

The variables defined at the current level determine the `element` numbers in the coarse and refined part of the mesh whereas the variables defined at next upper level determine sub elements corresponding to each element at the current level as long as the current level is not the highest level in the nested mesh hierarchy.
In the code snapshot, lines from 24 to 60 and from 65 to 90 show how sub elements corresponding to elements in the refined and coarse part of the mesh are determined and then feed to `struct EL` utilising a pre-defined *look-up table* and looping over sub elements respectively. This procedure, starts at the second highest level, it is then repeated over all mesh levels in a downward direction for all elements. On completion of this procedure, the setup of the

`struct EL` is completed and sub elements of an element at any given mesh level except the top level become accessible.

**Setup of struct EL with local mesh refinement**

```
for (lev=E->mesh.levmax-1; lev >= E->mesh.levmin; lev--) {
   xemin = E->info.ELXP[lev];
   zemin = E->info.ELZP[lev];
   yemin = E->info.ELYP[lev];

   xebnd = E->info.XEBND[lev];
   zebnd = E->info.ZEBND[lev];

   next = lev+1;
   xeminN = E->info.ELXP[next];
   zeminN = E->info.ELZP[next];
   yeminN = E->info.ELYP[next];

   xebndN = E->info.XEBND[next];
   zebndN = E->info.ZEBND[next];

   for (ye=1; ye <= yemin; ye++) {
      for (xe=1; xe <= xemin; xe++) {
         for (ze=1; ze <= zemin; ze++) {
            element = (ye-1)*zemin*xemin + (xe-1)*zemin + ze;

            /* for refined element corresponding to base mesh */
            if ( is_element_refined(E, lev, element) ) {
               sub_element = 2*(ye-1)*zeminN*xeminN + 2*(xe-1)*zeminN + (2*ze-1);

               for (elm=1; elm <= ends; elm++) {
                  element_count = ( element +
                                    (ye-1)*(zebnd/2)*(xebnd/2)*(ends-1) +
                                    (xe-1)*(zebnd/2)*(ends-1) + (ze-1) +
                                    offset[elm].vector[z] +
                                    offset[elm].vector[x]*zebnd +
                                    offset[elm].vector[y]*zebnd*xebnd/xemin );

                  sub_element_count = ( sub_element +
                                        offset[elm].vector[z] +
                                        offset[elm].vector[x]*zeminN +
                                        offset[elm].vector[y]*zeminN*xeminN );

                  for (yeN=1; yeN <= yeminN; yeN++) {
                     for (xeN=1; xeN <= xeminN; xeN++) {
                        for (zeN=1; zeN <= zeminN; zeN++) {
                           nxt_element_count = (yeN-1)*zeminN*xeminN + (xeN-1)*zeminN + zeN;

                           if ( nxt_element_count == sub_element_count ) {
                              for (el=1; el <= ends; el++) {
                                 E->EL[lev][element_count].sub[el] = ( sub_element_count +
                                                                       (yeN-1)*(zebndN/2)*(xebndN/2)*(ends-1) +
                                                                       (xeN-1)*(zebndN/2)*(ends-1) +
                                                                       (zeN-1) +
                                                                       offset[el].vector[z] +
                                                                       offset[el].vector[x]*zebndN +
                                                                       offset[el].vector[y]*zebndN*xebndN/xeminN );
                              }/* end of for el=1 */
                           }/* end of if nxt_element_count */

                        }/* end of for zeN=1 */
                     }/* end of for xeN=1 */
                  }/* end of for yeN=1 */

               }/* end of for elm=1 */
            }/* end of if ... */

            /* for coarse element corresponding to base mesh */
            else {
               element_count = ( element + (ye-1)*(zebnd/2)*(xebnd/2)*(ends-1) +
                                 (xe-1)*(zebnd/2)*(ends-1) +
                                 (zebnd/2)*(ends-1) );
               for (el=1; el <= ends; el++) {
                  sub_element_count = ( sub_element +
```

```
                                    offset[el].vector[z] +                                                          70
                                    offset[el].vector[x]*zeminN +                                                   71
                                    offset[el].vector[y]*zeminN*xeminN );                                           72
                                                                                                                    73
                  for (yeN=1; yeN <= yeminN; yeN++) {                                                               74
                     for (xeN=1; xeN <= xeminN; xeN++) {                                                            75
                        for (zeN=1; zeN <= zeminN; zeN++) {                                                         76
                           nxt_element_count = (yeN-1)*zeminN*xeminN + (xeN-1)*zeminN + zeN;                        77
                                                                                                                    78
                           if ( nxt_element_count == sub_element_count ) {                                          79
                              E->EL[lev][element_count].sub[el] = ( sub_element_count +                             80
                                                                    (yeN-1)*(zebndN/2)*(xebndN/2)*(ends-1) +        81
                                                                    (xeN-1)*(zebndN/2)*(ends-1) +                   82
                                                                    (zebndN/2)*(ends-1) );                          83
                           }/* end of if nxt_element_count */                                                       84
                                                                                                                    85
                        }/* end of for zeN=1 */                                                                     86
                     }/* end of for xeN=1 */                                                                       87
                  }/* end of for yeN=1 */                                                                         88
                                                                                                                    89
               }/* end of for el=1 */                                                                             90
            }/* end of else */                                                                                    91
                                                                                                                    92
         }/* end of for ze=1 */                                                                                  93
      }/* end of for xe=1 */                                                                                     94
   }/* end of for ye=1 */                                                                                        95
}/* end of for lev= ... */                                                                                       96
```

Setup of struct EL with local mesh refinement

Following on from the glimpse of a couple of C structures and a few code snapshots, Table 4.1 provides listing of the functions which are substantially modified to facilitate local mesh refinement. There is no intention to describe these modifications here due to the complexity involved and it wouldn't serve any useful purpose for this project. It is to be noted that functions with minor modifications are not included in this listing.

| Source File Name | Return Type | Function Name |
|---|---|---|
| Advection_diffusion.c | void | PG_timestep() |
| Boundary_conditions.c | void | velocity_boundary_conditions() |
| Boundary_conditions.c | void | temperature_boundary_conditions() |
| Boundary_conditions.c | void | velocity_refl_vert_bc() |
| Boundary_conditions.c | void | temperature_refl_vert_bc() |
| Boundary_conditions.c | void | temperature_imposed_botm_bcs() |
| Boundary_conditions.c | void | horizontal_bc() |
| Construct_arrays.c | void | construct_id() |
| Construct_arrays.c | void | construct_lm() |
| Construct_arrays.c | void | construct_node_maps() |
| Construct_arrays.c | void | construct_node_ks() |

. . . continue on page 57

. . . continued from page 56

| Source File Name | Return Type | Function Name |
|---|---|---|
| Construct_arrays.c | void | construct_masks() |
| Construct_arrays.c | void | construct_sub_element() |
| Construct_arrays.c | void | construct_elt_ks() |
| Construct_arrays.c | void | construct_elt_gs() |
| Convection.c | void | convection_initial_temperature() |
| Convection.c | void | convection_static_composition() |
| Convection.c | void | setup_plume_problem() |
| General_matrix_functions.c | int | solve_del2_u() |
| Global_operations.c | void | remove_horiz_ave() |
| Global_operations.c | void | return_horiz_ave() |
| Global_operations.c | float | return_bulk_value() |
| Global_operations.c | float | return_bulk_value_e() |
| Instructions.c | void | common_initial_fields() |
| Instructions.c | void | read_instructions() |
| Instructions.c | void | allocate_common_vars() |
| Instructions.c | void | global_derived_values() |
| Instructions.c | void | read_initial_settings() |
| Nodal_mesh.c | void | node_locations() |
| Parallel_related.c | void | parallel_domain_decomp() |
| Parallel_related.c | void | parallel_shuffle_ele_and_id() |
| Parallel_related.c | void | parallel_communication_routs() |
| Process_buoyancy.c | void | heat_flux() |
| Process_buoyancy.c | void | plume_buoyancy_flux() |
| Process_buoyancy.c | void | onsetssc() |
| Solver_multigrid.c | void | interp_vector() |
| Solver_multigrid.c | void | inject_node_fvector() |
| Tracers_init.c | void | trac_memalloc() |
| Viscosity_structures.c | void | get_system_viscosity() |

Table 4.1: List of substantially modified functions

In addition to the substantially modified functions, some new functions have been added to the CITCOM source code as well. A few of these function were added in phase 2 (previous phase) to add new functionality such as multi-

grid schemes whereas others aimed at making CITCOM more user friendly. However, most of these new functions, which are added here in phase 3, are related to local mesh refinement and a listing of these functions is given in Table 4.2 without any detail about implementation.

| Source File Name | Return Type | Function Name |
|---|---|---|
| Construct_arrays.c | void | construct_nei() |
| Construct_arrays.c | void | construct_sien() |
| Construct_arrays.c | void | construct_info_levmax() |
| Construct_arrays.c | void | construct_info() |
| Construct_arrays.c | int | is_element_refined() |
| Construct_arrays.c | int | int_compare() |
| Instructions.c | double | multigrid() |
| Instructions.c | double | sawtooth1_multigrid() |
| Instructions.c | double | sawtooth2_multigrid() |
| Nodal_mesh.c | int | inbox() |
| Output.c | void | date_and_time() |
| Output.c | void | data_directory() |
| Parallel_related.c | int | firstnode() |

Table 4.2: List of new added functions

### 4.2.4 Outcome

Within the existing framework of CITCOM, local mesh refinement is a difficult option to try. Finite elements used in CITCOM are regular-structured elements, squares or rectangles in two dimensions and cubes or cuboids in three dimensions. Each element has four sub elements in two dimensions and eight sub elements in three dimensions. A local refinement of one or more elements always creates *hanging* nodes on the edges of neighbouring elements which are not refined. In the case of three dimensions, these hanging nodes are also created on faces of neighbouring elements which are not refined. The number of element edges/faces with hanging nodes of non-refined elements depends on the surrounding number of neighbouring elements which are refined, e.g. one or more elements along one or more dimensions.
We have opted for the approach of a *one-level-difference* refinement rule

which is described in section 4.2.1. Due to the level of complexity involved in the refinement of a regular structured grid in two dimensions generally, and in three dimensions particularly, together with the complex structure of CITCOM itself this was not an easy task. However, we have managed to achieve partial success under these hard circumstances.
In CITCOM, there are a number of nested loops or cycles (not multigrid cycles) and within these loops CITCOM solves the given problem for the velocity field, temperature field, pressure field, viscosity field and tracers, etc. The computations from within one loop have the potential to impact on the computations within other loops. In other words, a specific part of the CITCOM code where multigrid schemes are implemented solves any given problem for the velocity field only but has the potential to get influenced by the computations carried out elsewhere in the code.
CITCOM solves time dependent problems but has the option to solve for a 0 (zero) time step case only. This behaviour is controlled from within the input file. For the case when CITCOM is restricted to the 0 (zero) time step, it could achieve a solution for a few small test problems (a smaller version of the test problems used in phase 2). We have tried this but it takes more time than expected after local mesh refinement. This could be due to the reason that elements of two different sizes, as a result of local mesh refinement, interface with each other without any smooth transition. In the case of time dependent problems, computations start deteriorating after only a few time steps. This behaviour is not understood but suspicion is that the advection-diffusion related computations may be the influencing factor. On the other hand, it is also possible that the non-multigrid part of the code, which performs advection-diffusion and tracers related computations may be in need of some more modifications to adjust for local mesh refinement.

# Chapter 5

# Conclusions

This chapter concludes the work undertaken under the dCSE project "Multigrid Improvements to CITCOM" presented in this report. A brief summary of the work undertaken in each of the three phases of this project is described in section 5.1 followed by achievements in section 5.2. This chapter is concluded with some recommendations aimed at further improvements to the CITCOM package.

## 5.1 Summary

After the introduction of this dCSE project in Chapter 1 describing the project duration, work plan and background of the CITCOM package, Chapter 2 describes the initial study undertaken towards understanding the CITCOM package and learning. Chapter 3 is dedicated to the description of Multigrid methods, a model problem, test problems, computational results and analysis of the results for representative test problems in two and three dimensions. These results are obtained using four multigrid schemes showing excellent scaling for each multigrid scheme. This is followed by a description of post processing tools used in preparing these results. Local mesh refinement strategy, setup and implementation followed by the outcome of this implementation is presented in Chapter 4. Chapter 5 concludes this project and report.

## 5.2 Achievements

The work carried out as part of this dCSE project enabled CITCOM to achieve faster convergence. For the best cases CITCOM performs *over 31% faster* for the V-cycle multigrid scheme, *over 38% faster* for the W-cycle multigrid scheme in comparison to the corresponding FMG(V) and FMG(W) schemes respectively for the simple 2D test problem and *over 12% faster* for the W-cycle multigrid scheme in comparison to the corresponding FMG(W) scheme for the complex 3D test problem.
Other observations based on the analysis of the four multigrid schemes along with tests problems and their results, in two and three dimensions, given in chapter 3 are summarised below. These observations account for the four multigrid schemes, namely, Multigrid V-cycle, Multigrid W-cycle, FMG(V) and FMG(W).

- For relatively simple and fast converging problems the Multigrid V-cycle scheme is the fastest scheme.
- Full Multigrid or FMG schemes perform well in contrast to the corresponding basic Multigrid V-cycle and Multigrid W-cycle schemes for complex and hard to solve problems. In these cases, basic schemes, namely, the Multigrid V-cycle and Multigrid W-cycle might fail to converge.
- V-cycle based multigrid schemes generally perform better than the corresponding W-cycle based multigrid schemes.
- The Multigrid V-cycle scheme offers optimal choice for relatively simple and easy to solve problems and the FMG(V) scheme offers optimal choice for relatively complex and hard to solve problems.
- Scaling of all multigrid schemes is generally excellent. This is particularly true if the sub problem size per MPI process is not reduced to the extremely smallest possible size, that is, just two elements per MPI process in any direction.
- The use of one or two cores per node instead of all four cores per node may slightly affect scaling, that is, the use of all cores per node gives best scaling. This is encouraging in the context of the efficient usage of multi-core configurations.

Local mesh refinement, within the existing framework of CITCOM, is a difficult option to try. In the presence of the high level of complexity involved in the refinement of a regularly structured grid in two dimensions generally, and in three dimensions particularly, together with the complex structure of CITCOM itself, we have managed to achieve partial success under these hard circumstances.
CITCOM solves time dependent problems but can be restricted to a 0 (zero) time step only. In the later case, it can achieve a solution for a few small test problems (smaller versions of the test problems used in phase 2). We have tried this approach but it takes more time than expected after local mesh refinement. It is thought to be the case as a result of the local mesh refinement. Here, there are elements of two different sizes which interface with each other without any smooth transition. In the former case of time dependence, computations start deteriorating after only a few time steps. This behaviour is not understood but suspicion is that the advection-diffusion related computation may be the influencing factor. On the other hand, it is also possible that the non-multigrid part of the code, which performs advection-diffusion and tracers related computations may be in need of some more modifications to adjust for the local mesh refinement.

## 5.3 Recommendations

This package has a potential for further improvements and a few of numerous possible ways in which it can be improved and extended are suggested here.

- Local mesh refinement needs to be addressed in the context of temperature field computations addressing advection-diffusion and convection related issues.
- Implementation of tracers needs to be re-visited in order of address the changes in the number of nodes and elements in the z-direction due to local mesh refinement in comparison to the global refinement to achieve the nested hierarchy of mesh levels.
- To get acceptable results for larger problems CITCOM requires a longer run time than the maximum (12 hrs) allowed on HECToR. The current option is to restart the CITCOM computation at the point of termination from last run where it had written information to the output

files essential for restarting. However, a vast amount of computation is performed again after restarting in some cases this may take up to *one-third* of the total run time after restart. If this can be modified to write the full information to the output files required for restarting without the need of repeating part of the computations, it could save a big chunk of time which is otherwise wasted in performing the same computation after each restart.

# Bibliography


[1] Auth, C. and Harder, H. Multigrid solution of convection problems with strongly variable viscosity. *Geophysical Journal International*, 137:793–804, 1999.

[2] Blankenbach, B. and et. al. A benchmark comparison for mantle convection codes. *Geophysical Journal International*, 98:23–38, 1989.

[3] Boyer, Franck, Lapuerta, Cline, Minjeaud, Sbastian, and Piar, Bruno. A local adaptive refinement method with multigrid preconditionning illustrated by multiphase flows simulations. volume 27, pages 15–53. INRIA a CCSD electronic archive server based on P.A.O.L [http://hal.inria.fr/oai/oai.php] (France), 2009. published on-line.

[4] Briggs, W., Henson, V., and McCormick, S. *A Multigrid Tutorial*. SIAM, 2000.

[5] CitcomS: A finite element code for thermal convection problems relevant to earths mantle. http://www.geodynamics.org/cig/software/packages/mc/citcoms/.

[6] Devillard, N. and Chudnovsky, V. Run-time function call tree with gcc. http://ndevilla.free.fr/etrace, March 8 2004.

[7] Dikov, V. Multigrid methods for solving differential equations. Technical report, Ferien Akademie, September 2005.

[8] Doxygen: Source code documentation generator tool. http://www.doxygen.org/.

[9] HECToR: UK national supercomputing service. http://www.hector.ac.uk/.

[10] Krysl, P., Grinspun, E., and Schröder, P. Natural hierarchical refinement for finite element methods. *Internat. J. Numer. Methods Engrg.*, 56(8):1109–1124, 2003.

[11] Moresi, L. and Gurnis, M. Constraints on the literal strength of slabs from three-dimensional dynamic flow models. *Earth Plan. Sci. Let.*, 138:15–28, 1996.

[12] Moresi, L. and Solomatov, V. S. Numerical investigation of 2d convection with extremely large viscosity variations. *Phys. Fluids*, 7(9):2154–2162, 1995.

[13] Gnuplot. http://www.gnuplot.info/.

[14] Wessel, P. and Smith, W. H. F. The generic mapping tools (gmt). http://gmt.soest.hawaii.edu/.

[15] Zhong, S., Zuber, M. T., Moresi, L. N., and Gurnis, M. The role of temperature dependent viscosity and surface plates in spherical shell models of mantle convection. *Journal of Geophysical Research*, 105(B5):11063–11082, 2000.