Stabilized Explicit Time Integration for Parallel Air Quality Models

BY

Anurag Srivastava

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University

Masters of Science
in
Computer Science and Applications

Dr Adrian Sandu, Chair

Dr Calvin J Ribbens

Dr Danesh Tafti

August 18, 2006
Blacksburg, Virginia

Keywords: Explicit Time-stepping, Air Quality Models,
Runge Kutta Chebyshev Integration

© Copyright by
Anurag Srivastava

Stabilized Explicit Time Integration for
Parallel Air Quality Models

Anurag Srivastava

(ABSTRACT)

Air Quality Models are defined for prediction and simulation of air pollutant concentrations over a certain period of time. The predictions can be used in setting limits for the emission levels of industrial facilities. The input data for the air quality models are very large and encompass various environmental conditions like wind speed, turbulence, temperature and cloud density.

Air Quality models are based on advection-diffusion equations. The differential equations used for such modeling are moderately stiff and require appropriate techniques for fast integration over large intervals of time. Implicit time stepping techniques for solving differential equations being unconditionally stable are considered appropriate for the solution. However, implicit time stepping techniques impose certain data dependencies that can cause the parallelization of air quality models to be inefficient.

The current approach uses Runge Kutta Chebyshev explicit method for solution of advection diffusion equations. It is found that even if the explicit method used is computationally more expensive in the serial execution, it takes lesser execution time when parallelized because of less complicated data dependencies presented by the explicit time-stepping. The implicit time-stepping on the other hand cannot be parallelized efficiently because of the inherent more complicated data dependencies.

ACKNOWLEDGMENT

The presented graduate work is only a small portion of what I have learned from Dr Sandu. It was through his encouragement that I pursued my interest in modeling and simulation to begin with, and it was further through his motivation and guidance, that I learned enough to contribute something to the ongoing work in the group. The project has been a great experience altogether, with just a little regret of not having learned as much as I possibly could have under the superb guidance that I had.

I also want to extend my thanks to Emil whose patient assistance accelerated the work through the porting and testing phase.

I cannot forget to mention my friend Raghavendra, whose company and enthusiasm has always kept me motivated for trying hard to achieve what I aspire.

TABLE OF CONTENTS

Page

ABSTRACT ii

ACKNOWLEDGEMENT iii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

1. INTRODUCTION 1
   - 1.1. Motivation 1
   - 1.2. Air Quality Models 8
2. THEORY 10
   - 2.1. Model Details 10
   - 2.2. Explicit Runge Kutta Chebyshev(RKC) method 18
3. PARALLELIZATION 23
   - 3.1. Background 23
   - 3.2. Parallelization of Air Quality Models 24
   - 3.3. Parallelization of Stem 31
4. IMPLEMENTATION 36
   - 4.1. Environment and Tools 36
   - 4.2. Libraries 36
   - 4.3. Modular Design of STEM 38
   - 4.4. Topology 40
   - 4.5. Code Flow 43
   - 4.6. Data Types used in the code 54
5. RESULTS AND DISCUSSIONS 70
   - 5.1. Test Problem 70
   - 5.2. Preliminary Tests for RKC Time Integration 70
   - 5.3. Results 73
   - 5.4. Software Configuration Management 80
   - 5.5. Design and Coding guidelines 81
6. CONCLUSIONS AND FUTURE WORK 92
   - 6.1. Conclusions 92
   - 6.2. Future Work 93

APPENDIX A. IMPLEMENTATION DETAILS 95

A.1. Stem Driver 96
A.2. Module communication 104
A.3. Module Computation 124

BIBLIOGRAPHY 136

VITA 138

LIST OF TABLES

Table Page

5.1 Time spent on anantham (seconds) for 1h simulation time 74

5.2 Time spent on system-X (seconds) for 1h simulation time 75

LIST OF FIGURES

Figure | Page

2.1 Eigenvalue of the Jacobian of the discrete advection-diffusion in directions x and y 21

2.2 Stability regions of Runge Kutta Chebyshev explicit time-stepping method with respect to s 22

3.1 Data Dependencies for Implicit and Explicit time-stepping 27

3.2 The domain of dependency (stencil) for 1-D spatial discretizations of the transport equation using Implicit and Explicit time-stepping schemes . . . 33

3.3 X and Y -slices in X-Y partitioning scheme 34

3.4 H and V -slices in H-V partitioning scheme 35

3.5 Tiled Partitioning 35

4.1 Modules in the program 39

4.2 The coordinate system for bookkeeping tiles- Every processor is mapped to a point in the system 42

4.3 The dimensions of the basic datatypes used in distribution 60

4.4 The need of ghost cells for the computation at every processor 61

4.5 The block at root is the north-west block in program's terminology 62

4.6 Exchange of data (ghost cells) at every stage 62

4.7 Blocks along boundary need limited exchange 63

4.8 Ghost Cells are appended to the matrix at every block 64

4.9 Enhanced matrix after appending the exchanged data from the neighbors . 65

4.10 Markers used in the program for calculation of derivative on the enhanced matrix after appending ghost cells 66

4.11 Assignment of block sizes 67

4.12 Distribution of input data (initial concentrations, wind fields and diffusion coefficients) to processors 68

4.13 Distribution of boundary data to respective processors 69

5.1 The grid-map of Asia - the region where the data were collected from . . . 70

5.2 The solution of advection diffusion equations at various times 71

5.3 The solution of advection diffusion equations at various times for numerical boundary conditions 72
5.4 Predicted vs Observed Execution times for distribution and computation . 82
5.5 Predicted vs Observed Execution times for exchange 83
5.6 Time spent on anantham 84
5.7 Time spent on system-X 85
5.8 Speedup on anantham 86
5.9 Speedup on system-X 87
5.10 Efficiency on anantham 88
5.11 Efficiency on system-X 89
5.12 Fractions of Execution Times - Implicit vs Explicit 90
5.13 Execution times for overlapped computation vs non-overlapped computation scheme 91

CHAPTER 1

INTRODUCTION

1.1 Motivation

The period between 14th century and the 17th century of human history marked the transition from what were termed the Dark Ages. It was for the first time in the history, when humans had started overcoming the fear that always prevented them from aspiring beyond what nature had given them. This new era influenced the human knowledge significantly. Through exploring, studying and exploring again, human beings had started to shun the ignorance that caused such fears.

The period in human history was aptly called the Renaissance, a movement that sowed the seeds of Science and Industrial revolution. By the 18th century human society was completely transformed, reflecting the changes in almost every aspect of life, and inculcating a completely different outlook towards nature and towards what lies beyond.

Slowly with time, human beings had learned to transform the environment to suit their needs, unlike till the dark ages, when they always submitted to the nature. Nature could hence be treated as another commodity in the new industrialized world. The debates on environmental ethics had thus only started to get intense.

With great power comes the great responsibility - so was it with the industrial might. By 20th century earth had started showing signs of degradation as a result of unchecked and irresponsible industrialization. Pollution of air and water had started to affect the ecosystem and consequently the normal human life.

Considering the atmosphere that surrounds use, the air has trace gases, carbon dioxide and water vapor that specifically play essential roles in climate. In the absence of these molecules that absorb strongly in the infrared, the surface temperature could become about 40 K colder

than it is today. At a temperature that low, the oceans would be frozen over, and life as we know it would be impossible. Unfortunately, if the pollution trends go unchecked such a destruction of life can become a reality.

1.1.1 The changes in atmosphere.

1.1.1.1 Carbon Dioxide.

The atmosphere has shown considerable changes because of the industrial emissions. The concentration of CO_2 in the Earth's atmosphere, for example, has risen steadily over the past 140 or so years, from about 280 parts per million in 1850 to about 350 parts per million. The change is mostly because of the combustion of fossil fuels. Since the Industrial Revolution, nearly 1.5×10^{11} metric tons of organic carbon have been mined and consumed in the form of coal, oil, and natural gas.

Approximately half of the carbon emitted since the Industrial Revolution persists in the atmosphere today. The balance is presumed to have made its way into the oceans or to have been incorporated into organic matter on land. The uptake of CO_2 by the oceans is limited by the supply of carbonate ions in surface waters. A continuing rise in carbon dioxide is inevitable. If current estimates for the reserve of fossil fuelsabout 4×10^{12} metric tons of carbonare considered, and if it is assumed that half of this reserve is used up over the next 100 years, the level of CO_2 could exceed 1,000 parts per million. If one assumes more conservatively that the consumption of fossil fuels will double over the next 100 years, CO_2 may be expected to grow to about 600 parts per millionapproximately twice the value in 1850.

1.1.1.2 Ozone.

To consider ozone depletion as another instance, it was not before 1970 that scientists began to focus on the fact that even small changes in O_3 can have a significant impact on humans.

Investigators observed that migration of people to lower latitudes, the shift in population from the northeastern part of the United States to the Sun Belt for example, was accompanied by an alarming rise in the incidence of skin cancer.

Much of the work undertaken since the mid-1970s has focused on the effects of CFC's on the assumption that the composition of the atmosphere was otherwise constant. It has become clear, however, that the response of ozone depends not simply on the abundance of CFC's but also on the abundances of methane, nitrous oxide, and carbon monoxide. These species, too, are changing. Current models suggest that a continuing release of CFC's at the rate registered in 1980, other gases remaining constant, would lead to a reduction in stratospheric ozone by about 5 percent. Maximum impact is predicted to occur at altitudes above 25 kilometers. An increase in nitrous oxide of 20 percent is expected to cause a reduction in ozone of about 2 percent. An increase in carbon dioxide should lead to a reduction in stratospheric temperatures with a consequent reduction in the anticipated impact of CFC's and nitrous oxide on ozone. The effect of an increasing burden of methane is more complex. Oxidation of methane provides a source of ozone at low altitude, while the reaction of chlorine with methane converts chlorine radicals to hydrogen chloride, resulting in a reduction in the impact of CFC's between 30 and 40 kilometers.

However, it is predicted by the models of pollution, that the change in the column density of stratospheric ozone to date would be relatively small. Reductions in ozone at high altitude, near 40 kilometers, ought to be balanced by excess production at low altitudes due in part to the higher level of methane and in part to NO_x released by high-altitude aircraft. Observational evidence is consistent with this view. A statistical analysis concluded that the change in the ozone column from 1970 to 1983 averaged -0.003 percent per decade. Such a foresight is indeed desirable for decision-making as policies for the industrial practices.

1.1.1.3 Trace gases.

Considering the trace gases F-11, F-12, methane, and nitrous oxide next, these happen to be even more potent towards global warming on a molecule-per-molecule basis. It has been estimated that a molecule of F-11 or F-12, for example, can produce warming equivalent to that caused by 104 molecules of CO_2. Perturbations to stratospheric ozone and tropospheric ozone also can have a significant impact. In short, a comprehensive theory of climate change must allow for all of these various changes in atmospheric composition.

1.1.2 The need for atmospheric models.

1.1.2.1 Long term predictions.

Most assessments of climatic change are based on simple, one-dimensional energy-balance models. These models provide an estimate for the globally averaged change in surface temperature. Studies suggest that the rise in carbon dioxide since 1850 should have resulted in an increase in the contemporary surface temperature of about 0.4 C. The trace gases F-11, F-12, methane, nitrous oxide, ozone, and stratospheric water vapor are estimated to have added to heating by CO_2 an increment equivalent of an additional 0.16 C, for a net change of 0.56 C. A change of this magnitude is generally consistent with observation.

The effects of trace gases are expected to grow relative to CO_2 in the years ahead. According to results from one particular simulation, CO_2 was predicted to cause an increase in surface temperature of about 0.9 deg. Celsius by the year 2030. With other trace gases included, it was estimated that the temperature increase could be as large as 2.2 deg. Celsius. A change of this magnitude would be unprecedented in recent Earth history.

Such long term predictions are necessary to assess the methods we use for harnessing energy sources. A healthy ecological environment cannot be compromised for short term goals of fulfillment of energy at a certain level. A comprehensive assessment requires a model

incorporating all of the important feedbacks between the atmosphere, oceans, and biosphere, with spatial resolution sufficient to distinguish at least the major biomes. Preliminary studies with three-dimensional models similar to those used for predicting weather suggest that climatic zones can be expected to shift to higher latitudes on a warmer Earth. Further work is required, however, before models of this kind can be used with confidence for quantitative planning.

1.1.2.2 Public Policy.

The public policy has appropriately reflected the concern over pollution. In the olden days of England in the 1830s, for example, excessive pollution was discouraged through the law. The governments of the civilized world have always been prompt in taking right measures to counter the threats that environmental pollution poses.

In more recent times, the United States Congress passed the Clean Air Act in 1963, the Clean Air Act Amendment in 1966, the Clean Air Act Extension in 1970, and Clean Air Act Amendments in 1977 and 1990. Numerous state governments and local governments have enacted similar legislation, either implementing federal programs or filling in locally important gaps in federal programs.

Clear Skies Act

The Clear Skies policy was put together by Jim Connaughton, Chairman of the Council on Environmental Quality, and was declared on February 14, 2002. The initiative is based on the idea "that economic growth is key to environmental progress, because it is growth that provides the resources for investment in clean technologies." The resulting proposal was a market-based cap-and-trade approach which intends to legislate power plant emissions caps without specifying the specific methods used to reach those caps.

Current power plant emissions amounted to 67% of all sulfur dioxide (SO_2) emissions (in the

United States), 37% of mercury emissions, and 25% of all nitrogen oxide (NO_x) emissions. Only SO_2 has been administered under a cap-and-trade program.

The goals of the Initiative are three-fold:

- Cut SO_2 emissions by 73%, from emissions of 11 million tons to a cap of 4.5 million tons in 2010, and 3 million tons in 2018.
- Cut NO_x emissions by 67%, from emissions of 5 million tons to a cap of 2.1 million tons in 2008, and to 1.7 million tons in 2018.
- Cut mercury emissions by 69%, from emissions of 48 tons to a cap of 26 tons in 2010, and 15 tons in 2018.

Actual emissions caps would be set to account for different air quality needs in the East and West.

Through the use of a market-based cap-and-trade program, the intent of the Initiative was to reward innovation, reduce costs, and guarantee results. Each power plant facility is required to have a permit for each ton of pollution emitted. Because the permits are tradeable, companies would have a financial incentive to cut back their emissions using newer technologies. This Initiative was modeled on the successful SO_2 emissions trading program in effect since 1995.

1.1.2.3 Air Quality Indices- Controlled Pollution.

The Air Quality Index (AQI) is a standardized indicator of the air quality in a given location. It measures mainly ground-level ozone and particulates except the pollen count, but may also include sulfur dioxide, and nitrogen dioxide. Various agencies around the world measure such indices, though definitions may change between places [20].

Health classifications used by the Environmental Protection Agency

- 0-50: Good (green)
- 51-100: Moderate (yellow)
- 101-150: Unhealthy for sensitive groups (orange)
- 151-200: Unhealthy (red)
- 201-300: Very unhealthy (purple)
- 301-500: Hazardous (maroon)

(AQI 100 corresponds to 0.08 ppm ozone; other levels for other pollutants)

The AQI can worsen (go up) due to lack of dilution with fresh air. Air, often caused by an anticyclone or temperature inversion, or other lack of winds lets air pollution remain in a local area. On these days, the media may ask the public to carpool or use public transport, or take other air pollution prevention measures.

The Met Office of the United Kingdom (UK) issues air quality forecasts wherein the level of pollution is described either as an index (ranging from 1 to 10) or as a banding (low, moderate, high or very high). These levels are based on the health effects of each pollutant.

The Air Pollution Index (API) levels for Hong Kong are related to the measured concentrations of ambient respirable suspended particulate (RSP), sulfur dioxide (SO_2), carbon monoxide (CO), ozone (O_3) and nitrogen dioxide (NO_2) over a 24-hour period based on the potential health effects of air pollutants.

An API level at or below 100 means that the pollutant levels are in the satisfactory range over 24 hour period and pose no acute or immediate health effects. However, air pollution

consistently at “High” levels (API of 51 to 100) in a year may mean that the annual Hong Kong “Air Quality Objectives” for protecting long-term health effects could be violated. Therefore, chronic health effects may be observed if one is persistently exposed to an API of 51 to 100 for a long time.

“Very High” levels (API in excess of 100) means that levels of one or more pollutant(s) is/are in the unhealthy range. The Hong Kong Environmental Protection Department provides advice to the public regarding precautionary actions to take for such levels [20].

1.2 Air Quality Models

Air quality models involve simulation of the dispersion of air pollutants in the ambient atmosphere. Dispersion models are used to estimate or to predict the downwind concentration of air pollutants emitted from sources such as industrial plants and vehicular traffic. The models are important to governmental agencies tasked with protecting and managing the ambient air quality. The models are can be used to determine whether existing or proposed new industrial facilities are or will be in compliance with national ambient air quality standards. The models also serve to assist in the design of effective control strategies to reduce emissions of harmful air pollutants.

The dispersion models require the input of data which includes:

- Meteorological conditions such as wind speed and direction, the amount of atmospheric turbulence (as characterized by what is called the “stability class”), the ambient air temperature and the height to the bottom of any inversion aloft that may be present.
- Emissions parameters such as source location and height, source vent stack diameter and exit velocity, exit temperature and mass flow rate.
- Terrain elevations at the source location and at the receptor location.

- The location, height and width of any obstructions (such as buildings or other structures) in the path of the emitted gaseous plume.

Many of the modern, advanced dispersion modeling programs include a pre-processor module for the input of meteorological and other data, and many also include a post-processor module for graphing the output data and/or plotting the area impacted by the air pollutants on maps.

The atmospheric dispersion models are also known as atmospheric diffusion models, air dispersion models, air quality models, and air pollution dispersion models.

Air quality models generally work on a discrete grid covering the atmospheric region to be simulated. The goal of the air quality models is to predict the concentrations of the pollutants after a certain period of time. The simulation times are usually very long, and therefore, one has to make sure that the appropriate techniques are applied. Parallelizability becomes very desirable, since the computations are really intensive. In the AQMs that were surveyed in this work, the phenomena involved in air pollution were modeled through differential equations.

Variational methods (3D-var 4D-var) provide an optimal control approach to the data assimilation problem. The success of the data assimilation depends on various factors like availability of data, accuracy of the background estimates, assimilation window, errors in measurements and errors in model representation [7].

CHAPTER 2

THEORY

2.1 Model Details

In a domain Ω, let n be the outward normal vector on each point of the boundary $\delta\Omega$. At each moment the boundary of the domain is partitioned into $\delta\Omega = \Gamma^I U \Gamma^O U \Gamma^G$ where $\Gamma^G$is the ground level portion of the boundary; Γ^I is set of boundary points where $u * n \leq 0$ and Γ^O the set where $u \cdot n > 0$

The evolution of concentrations in time is described by the material balance equations [7]:

$$\frac{\partial c_i}{\partial t} = -u\nabla c_i + \frac{1}{\rho}\nabla(\rho K \nabla c_i) + \frac{1}{\rho} f_i(\rho c) + E_i, t^0 \leq t \leq T \tag{2.1}$$

$$c_i(t^0, x) = c_i^0(x) \tag{2.2}$$

$$K\frac{\partial c_i}{\partial n} = 0 for x \epsilon \Gamma^O \tag{2.3}$$

$$K\frac{\partial c_i}{\partial n} = V_i^{dep} c_i - Q_i \text{ for } x\epsilon\Gamma^G, \text{ for all } 1 \leq i \leq s \tag{2.4}$$

Here,

c - concentration

ρ - air density

K - turbulent diffusivity tensor

V_i^{dep} - deposition velocity of species i

u - horizontal wind velocity

f_i - rate of chemical transformations

This system can be solved by a sequence of N time-steps of length Δt between t^0 and $t^N = T$. At each time step the approximation $c^k(x) = c(t^k, x)$ is applied at $t^k = t^0 + k\Delta t$

$c^{k+1} = N_{t^k,t^{k+1}} c^k$

$$N_{t^k,t^{k+1}} c_k = T_{xy}^{\Delta t} T_z^{\Delta t} C^{\Delta t} T_z^{\Delta t} T_{xy}^{\Delta t} \tag{2.5}$$

Here,

N- numerical operator

T_{xy}- two dimensional integrator for transport

C - integrator for chemistry

2.1.1 Horizontal transport.

The two dimensional derivative involved in the transport balance equations is evaluated as follows:

$$\frac{\delta c}{\delta t} = -u(x,y)\frac{\delta c}{\delta x} - v(x,y)\frac{\delta c}{\delta y} + \frac{1}{\rho}\frac{\delta}{\delta x}\rho K\frac{\delta c}{\delta x} + \frac{1}{\rho}\frac{\delta}{\delta y}(\rho K\frac{\delta c}{\delta y}) \tag{2.6}$$

where,

$c(t, x_I) = c_I(t)$ (inflow boundary)

$K\frac{\delta c}{\delta x}\mid_{x_{out}} = 0$ (outflow boundary)

Here,

u - Horizontal velocity component, along x

v - Horizontal velocity component, along y

The first and second terms in (2.6) are the advection terms, while the third and fourth terms are the diffusion terms.

Using finite differences to solve the system, the advection terms can be approximated as [7]

$$-u(x,y)\frac{\delta c}{\delta x}\mid_{x=x_i,y=y_j}=\begin{cases} u(x,y)\frac{(-c_{i-2,j}+6c_{i-1,j}-3c_{i,j}-2c_{i+1,j})}{6\Delta x}, \text{ if } u(x,y)\geq 0 \\ u(x,y)\frac{(2c_{i-1,j}+3c_{i,j}-6c_{i+1,j}+c_{i+2,j})}{6\Delta x}, \text{ if } u(x,y)<0 \end{cases} \tag{2.7}$$

Similarly,

$$-v(x,y)\frac{\delta c}{\delta y}\mid_{x=x_i,y=y_j}=\begin{cases} v(x,y)\frac{-c_{i,j-2}+6c_{i,j-1}-3c_{i,j}-2c_{i,j+1}}{6\Delta y}, \text{ if } v(x,y)\geq 0 \\ v(x,y)\frac{(2c_{i,j-1}+3c_{i,j}-6c_{i,j+1}+c_{i,j+2})}{6\Delta y}, \text{ if } v(x,y)<0 \end{cases} \tag{2.8}$$

For the diffusion terms,

$$\frac{1}{\rho}\frac{\delta}{\delta x}(\rho K\frac{\delta c}{\delta x})\mid_{x=x_i,y=y_j}=A_1-A_2 \tag{2.9}$$

where,

$$A_1=\frac{(\rho_{i+1,j}K_{i+1,j}+\rho_{i,j}K_{i,j})(c_{i+1,j}-c_{i,j})}{2\rho_1\Delta x^2}$$

$$A_2 = \frac{(\rho_{i,j}K_{i,j} + \rho_{i-1,j}K_{i-1,j})(c_{i,j} - c_{i-1,j})}{2\rho_1 \Delta x^2}$$

Similarly,

$$\frac{1}{\rho}\frac{\delta}{\delta y}(\rho K \frac{\delta c}{\delta y}) \mid_{x=x_i, y=y_j} = A'_1 - A'_2 \tag{2.10}$$

where,

$$A'_1 = \frac{(\rho_{i,j+1}K_{i,j+1} + \rho_{i,j}K_{i,j})(c_{i,j+1} - c_{i,j})}{2\rho_1 \Delta y^2}$$

$$A'_2 = \frac{(\rho_{i,j}K_{i,j} + \rho_{i,j-1}K_{i,j-1})(c_{i,j} - c_{i,j-1})}{2\rho_1 \Delta y^2}$$

2.1.2 Vertical Transport.

The equations for vertical transport require only the z-axis:

$$-w(x,y)\frac{\delta c}{\delta z} \mid_{z=z_k} = \begin{cases} -w(x,y)\frac{(c_k - c_{k-1})}{(z_k - z_{k-1})}, \text{ if } w_k \geq 0 \\ -w(x,y)\frac{(c_{k+1} - c_k)}{(z_{k+1} - z_k)}, \text{ if } w_k < 0 \end{cases} \tag{2.11}$$

2.1.3 Chemistry.

Chemistry involves ordinary differential equations that are very stiff, and need an implicit method to be solved. The current approach doesn't change the chemistry related computations, because the latter don't influence the communication among processors at all.

2.1.4 Boundary Adjustment.

For calculation of spatial derivatives at the grid points close to the boundary, an additional term needs to be added to account for the boundary condition.

Let $B_x(1,:)$, $B_x(2,:)$ be the boundary adjustment terms for the points along first boundary along x direction, and $B_x(n_x-1,:)$, $B_x(n_x,:)$ be the boundary adjustment terms for the points along the second boundary along x-direction.

$$B_x(1,j) = \frac{(C_x^B(1,j)u_{1,j} + 2Kh_{1,j}C_x^B(1,j))}{2\Delta x^2}, \text{ if } u_{1,j} \geq 0$$

$$B_x(2,j) = -1/6\frac{(C_x^B(1,j)u_{2,j})}{\Delta x}, \text{ if } u_{2,j} \geq 0$$

$$B_x(n_x-1,j) = \frac{(C_x^B(2,j)u_{n_x-1,j})}{6\Delta x}, \text{ if } u_{n_x-1,y} < 0$$

$$B_x(n_x,j) = -\frac{u_{n_x,j}C_x^B(2,j)}{\Delta x} + 2\frac{Kh_{n_x,j}C_x^B(2,j)}{2\Delta x^2}, \text{ if } u_{n_x,y} < 0$$

where, C_x^B is the boundary data for x-direction.

Similarly, let $B_y(:,1)$, $B_y(:,2)$ be the boundary adjustment terms for the points along first boundary along y-direction, and $B_y(:,n_y-1)$, $B_y(:,n_y)$ be the boundary adjustment terms for the points along the second boundary along y-direction, then

$$B_y(i,1) = \frac{(C_y^B(1,i)v_{i,1} + 2Khi, 1C_y^B(1,i))}{2\Delta y^2}, \text{ if } v_{x,1} \geq 0$$

$$B_y(i,2) = -1/6\frac{(C_y^B(1,i)v_{i,2})}{\Delta y}, \text{ if } v_{x,2} \geq 0$$

$$B_y(i,n-1) = \frac{(C_y^B(2,i)v_{i,n_y-1})}{6\Delta y}, \text{ if } v_{x,n_y-1} < 0$$

$$B_y(i,n_y) = -\frac{(v_{i,n_y}C_y^B(2,i)}{\Delta x} + 2\frac{(Kh_{i,n_y}C_y^B(2,i))}{2\Delta y^2}, \text{ if } v_{x,n_y} < 0$$

where, C_y^B is the boundary data for y-direction.

2.1.5 Integration of Differential Equations.

The time integration of differential equations is done either using explicit methods or the implicit methods. With explicit methods the current state is dependent only on the previous states. A typical explicit method can be mathematically represented as

$$x_{n+1} = f(x_n, x_{n-1}, ...)$$

The Euler method, for example, is an explicit method. In an implicit method on the other hand, the current state may not be dependent only on the previous states. An implicit method can mathematically be represented as

$$x_{n+1} = f(x_{n+1}, x_n, x_{n-1}, ...)$$

The implicit class of methods provide unconditionally stability with the disadvantage that such methods might involve considerable amount of calculations.

With stiff problems, implicit methods are necessary due to presence of severe time-step restrictions in the problems. The implicit methods are what we use in the current implementations of STEM. However, as would be explained further, the implicit methods hinder the parallelization by posing intricate data dependencies. If the problems are moderately stiff and are incurred by explicit methods certain explicit methods have sufficient stability. Verwer et al. [2],[3] report that Runge Kutta Chebyshev methods show extended stability domains along the negative real axis.

The Eigenvalue plot of the Jacobian matrix are shown in Figure 2.1. The moderately large eigenvalues on the negative real axis (See Figure 2.1) arise due to the atmospheric diffusion process (discretized in space).

Our aim was to look for explicit methods that have good stability property for eigenvalues shown in Figure 2.1. In other words, the eigenvalues of the Jacobian should lie under the

stability region(s) of the explicit method to be used. Given the distribution of eigenvalues, we found that with slight adjustments of the Runge Kutta Chebyshev methods, one can obtain a stable finite difference solution to the advection-diffusion system described in Equations 2.6 without reducing the time-step to a very small value. With the RKC methods (further discussed) the system was forwarded with time-steps as large as 15 minutes (See Figure 2.2 for stability regions of RKC methods used).

2.2 Explicit Runge Kutta Chebyshev(RKC) method

For an Ordinary Differential Equation (ODE) system,

$$w'(t) = F(t, w(t)),\ t > 0, w(0) = w_0 \tag{2.12}$$

The second order explicit Runge Kutta Chebyshev method [2] can be stated as:

$$W_0 = w_n$$

$$W_1 = W_0 + \bar{\mu_1} F_0$$

$$W_j = (1 - \mu_j - \nu_j)W_0 + \mu_j W_{j-1} + \nu_j W_{j-2} + \bar{\mu_j}\tau F_{j-1} + \bar{\gamma_j}\tau F_0 \tag{2.13}$$

$$w_{n+1} = W_s$$

where, $j = 2...s$ and $F_k = F(t_n - c_k\tau, W_k)$

w_n = solution vector (concentrations) at t_n

w_j = internal stages of the method at every iteration

The parameters for the formula are set so that in the relation $W_j = P_j(z)W_0$ at each stage of the iteration. Hence, the stability function $P_j(z)$ satisfies the following criteria [2],

1. nearly optimal stability of $P_s(z)$ for parabolic problems
2. internal stability
3. second order consistency of all $W_j (2 \leq j \leq s)$

This is achieved by using the first kind Chebyshev polynomial, satisfying the following three-term recurrence relation

$$T_j(x) = 2xT_{j-1}(x) - T_{j-2}(x), j = 2, 3, ..., s \tag{2.14}$$

where, $T_0(x) = 1$ $T_1(x) = x$

Hence $P_j(z)$ (in the relation $W_j = P_j(z)W_0$) becomes

$$P_j(z) = a_j + b_j T_j(\omega_0 + \omega_1 z) \tag{2.15}$$

where,

$$a_j = 1 - b_j T_j(\omega_0)$$

$b_0 = b_2$

$b_1 = \frac{1}{\omega_0}$ $b_j = T_j''(\omega_0)/(T_j'(\omega_0))^2, j = 2...s$

with,

$\omega_0 = 1 + \epsilon/s^2$ $\omega_1 = T_s'(\omega_0)/T_s''(\omega_0)$

(ϵ was chosen to be 2/13 in the program.)

The constants in the equation. (2.13) are,

$$\bar{\mu_1} = b_1\omega_1$$

for $j = 2...s$

$$\mu_j = \frac{2b_j\omega_0}{b_{j-1}}$$

$$\nu_j = -\frac{b_j}{b_{j-2}}$$

$$\bar{\mu_j} = \frac{2b_j\omega_1}{b_{j-1}}$$

$$\bar{\gamma}_j = -a_{j-1}\bar{\mu}_j$$

In the current work RKC algorithms were used with $s = 2, 3$ and 5. The stability function $P_s(z)$ for the values of s are:

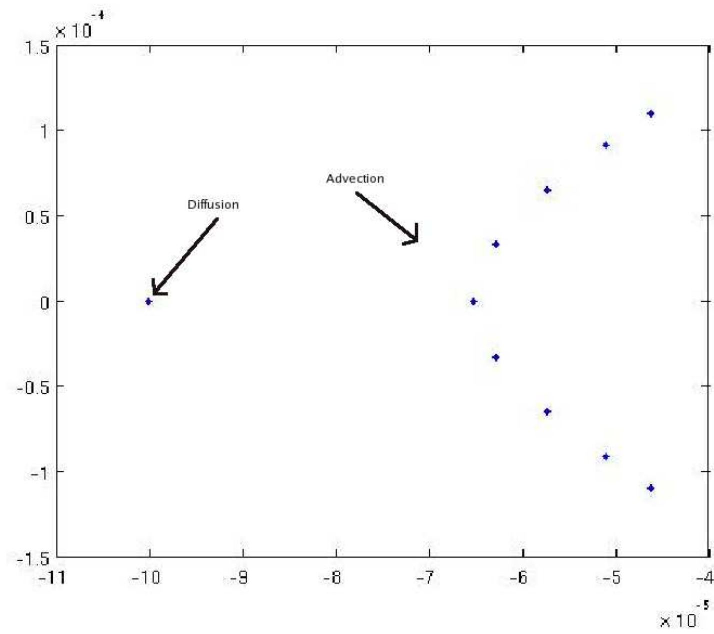
$$P_2(z) = \frac{899}{1681} + \frac{676}{1681}\{2(\frac{27}{26} + \frac{41}{52}z)^2 - 1\}$$

$$P_3(z) = \frac{51427}{78400} + \frac{19773}{78400}\{4(\frac{140}{351}z + \frac{27}{26})^3 - 3(\frac{140}{351}z + \frac{27}{26})\}$$

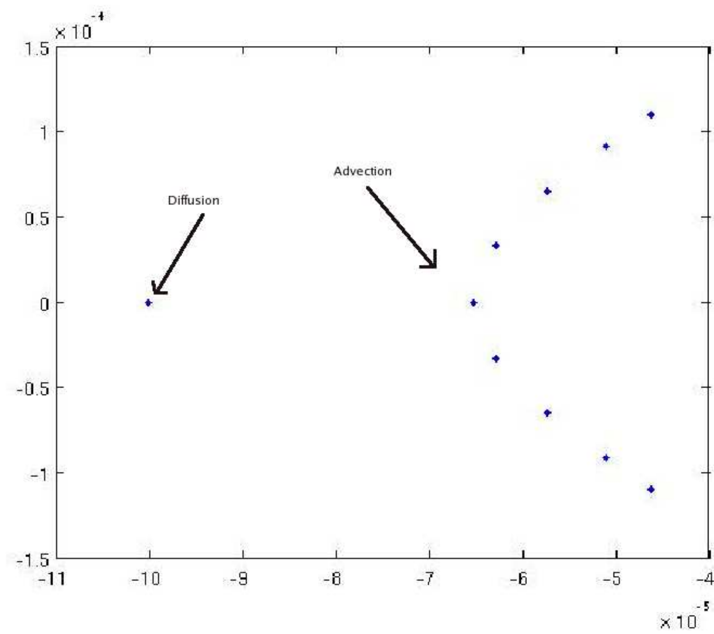
$$P_5(z) = .6552 + .2979\{5(1.0062 + .1279z) - 20(1.0062 + .1279z)^3 + 16(1.0062 + .1279z)^5\}$$

The stability region of the RKC methods used in the program are shown in Figure 2.2. As can be seen, the regions of stability include the negative part of the complex plan where the eigenvalues of the transport Jacobian reside.

The linear stability theory tells us that RKC methods can be used on transport equations with a large step size. From the stability regions shown in Figure 2.2 it can be inferred that the allowable time-steps would be 10^4-10^5. We have used time-steps of size 15 minutes (300s) in the program.

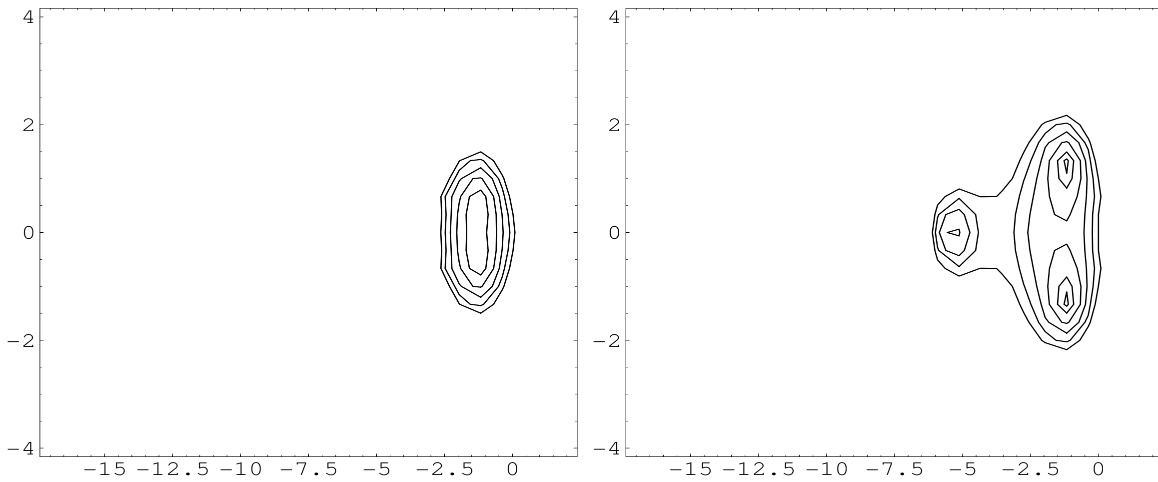


(a) Jacobian J_x



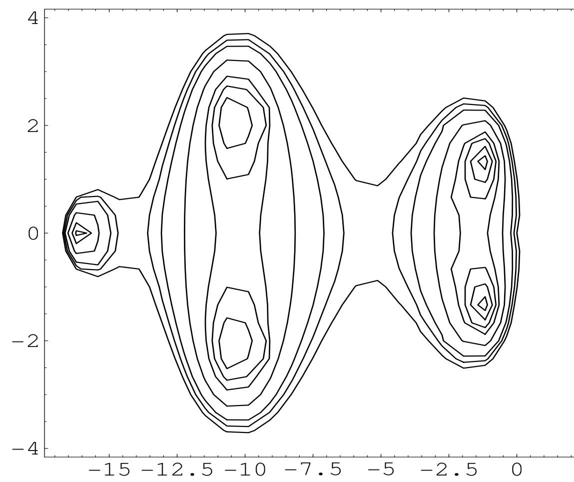
(b) Jacobian J_y

Figure 2.1. Eigenvalue of the Jacobian of the discrete advection-diffusion in directions x and y



(a) RKC, s=2

(b) RKC, s=3



(c) RKC, s=5

Figure 2.2. Stability regions of Runge Kutta Chebyshev explicit time-stepping method with respect to s

CHAPTER 3

PARALLELIZATION

3.1 Background

The first instance of parallelization as in idea of computation might have been first exhibited by IBM's 704 back in 1955. But parallelization didn't become popular until late 70s with the advent of VLSI.

The idea of cluster computing was introduced later in 1994 (Beowulf clusters) It was demonstrated that a parallel computer with reasonable computational ability could be built out of personal computer hardware. This achievement spawned development of many low-cost cluster-based computing machines. It didn't take long before the high-performance computing research community embraced the Beowulf philosophy.

Attempts have been made to let computer programs find out the optimal strategy for distribution of task to processors. The research goal here, was to abstract the parallelization from the programmer completely through a parallel compiler. However, the results have not been very satisfactory. A parallel language has more or less, been the choice for the research community, i.e. a language that only facilitates parallelization for the programmer yet giving the freedom to parallelize the parts of the program to the programmer, and leaving the responsibility of designing the parallel program absolutely on the programmer.

Our current approach is along the same lines, i.e. designing parallelization beforehand, and optimizing the application for the particular environment. The scope of parallelization was analyzed, and the various strategies for parallelization were assessed before implementing the solution.

In parallelized applications, it is observed that with increasing number of processors, the execution time doesn't decrease linearly. There is what can be termed as the 'parallelizability'

of an application, the extent to which the application can be feasibly parallelized. After a certain threshold, the speedup achieved out of parallelization even starts to decrease with increasing the number of processors. A badly parallelized application may not scale to more than a few number of processors. The gain we obtain from such a parallelization, therefore, is not too much.

The metrics that assess how 'good' a parallelization scheme is, are described later along the analysis of results from the current work.

3.2 Parallelization of Air Quality Models

3.2.1 Previous Work.

Elbern [9] examined contribution of physical-chemical and dynamic modules to load imbalances, working on the EURAD (European Air Pollution Dispersion Model). He termed what are known as local and global decision bases. In a local decision base, the work load of a processor is compared with only the immediate neighbors of the processor in the topology, while in a global decision base, the load of all the processors involved has to be considered. Elbern chose four grid partitioning strategies to balance the load:

1. Equal Area Partitioning - EAP - A static scheme where each processor maintains the same amount of non-neighboring grid cells

2. Static Load Balancing - SLB - Involves arrangement of grid rows and columns depending on the prior known distribution of computational work once before the simulation.

3. One dimensional dynamic load balancing - DL1 - Involves changing the row and column positions after each time step depending on the computational load at every processor. The decision is made based on slow temporal variances of the load, and the detection of regular imbalance patterns. (e.g. during dusk and dawn solar radiation changes

enforces small chemical time steps)

4. Two dimensional dynamic load balancing - DL2 - This is same as the DL1 with the primary difference being the ability of each processor to adjust its north and south bounds separately.

It was concluded from the experiments conducted with the above partitioning strategies, that the full adjustment scheme (i.e. the two dimensional load balancing) produced the best results. The author concluded that full processor use without any idle time cannot be achieved due to high number of different complex tasks at each step.

Dabdub et.al.[11] ported a parallel implementation of AQMs on MIMD machines and presented a performance model to predict execution of times of a program using some machine-dependent and application-dependent parameters. For the California Institute of Technology photochemical model run on 12 different machines, the authors report that 87% of the total time is consumed by chemical and vertical transport computations, while input and output routines for the large data files account for 7.5% of the computation. Only 5% of the total time was spent on horizontal computation. Dabdub et. al. used a master-slave strategy to parallelize the AQM, where the master performed all the I/O along with the distribution of the work to rest of the processors (slaves).

In the performance model that Dabdub et. al.[11] present, the load imbalances are ignored, and the total execution time is estimated only with the communication time and computation time. This may result in significant deviation from the observed execution times.

W. Owczarz and Z. Zlatev [8] parallelized the Danish Eulerian Model (DEM) on an IBM SMP machine, utilizing the machine dependent optimizations along with the general optimizations. The IBM SMP used for the experiments was a 16 processor machine, with a pair of 8 processors on two nodes. The 8 processors on every node had shared memory with there

being a distributed memory between the two nodes. The optimization techniques involved:

- caching
- reversing of (DO and FOR loops)

Such optimizations were reported to be system dependent. As an example, the vector machines would perform badly when using blocking techniques for the simple reason that vector machines are built to handle larger array more efficiently. A technique not utilizing the fact would result in poor performance.

Also, since the communication overhead for the distributed memory is much higher, the program minimized the communication through distributed memory, making most of the communication through the shared memory between the eight processors in the pair. With all the optimizations, the serial code reported the speedup of 3.7. When parallelized, the authors obtained a speedup of 6.54 on one node (82% efficiency) and that of 12.32 (72% efficiency) on two nodes.

Miehe et al. [6] developed an H-V communication library reducing the computation time significantly for air quality models using H-V partitioning and reshuffling. The H-V partitioning was reported to scale the application efficiently to larger number of processors. The efficiency of the parallel application was above 50% on about 60 processors.

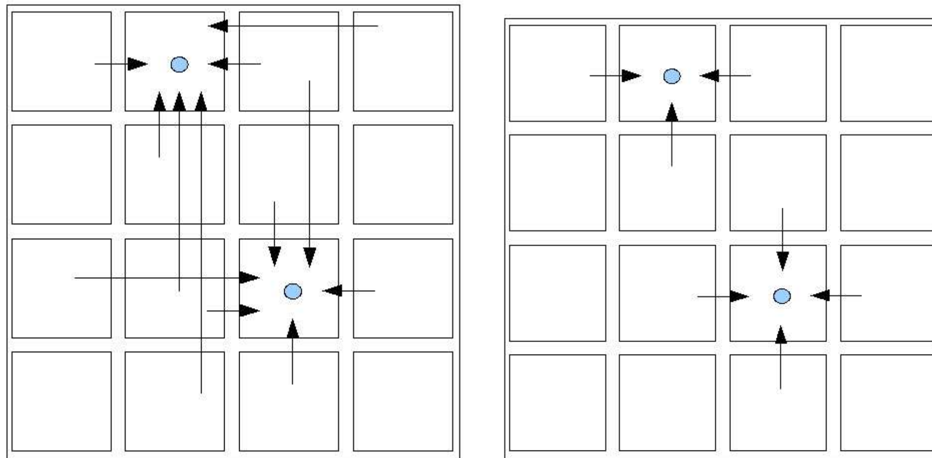
3.2.2 Data Dependencies in Implicit and Explicit methods.

If choosing an implicit time-stepping for the transport equations, the integration starts from the boundary point with the initial conditions, and iterates through the final boundary point. In this manner the whole data along the corresponding dimension (along which the

integration takes place) is needed. What this means for the two-dimensional case (i.e. when a two-dimensional derivative being used in the integration) is that the whole matrix is needed at every sub-domain in the integration scheme.

Quite reasonably, this is a reason strong enough to look for an alternative way of partitioning (other than the X-Y partitioning) when using the implicit method. A better method reported for this case, is H-V partitioning (to be further discussed in detail)

When using an explicit method however, only a window of the data from the previous state is required. For our two-dimensional case this means that we only need to have data from the neighboring blocks (explained in figure) This being an explicit method, doesn't make the technique unconditionally stable. Since data is to be exchanged at multiple stages in every iteration, the communication is not too significantly reduced (Figure 3.2,3.1).



(a) Data dependencies in a method with Implicit time-stepping

(b) Data dependencies in a method with Explicit time-stepping

Figure 3.1. Data Dependencies for Implicit and Explicit time-stepping

3.2.3 Partitioning and Slicing strategies.

As discussed in the previous section, the different data dependencies are handled by appropriate partitioning strategies. The aim of a partitioning strategy is to minimize the communication between processors both in terms of the total data to be transmitted and the number of times the data has to be transmitted.

The main data structure that is used in STEM is the 4-D concentration array of size $nx \times ny \times nz \times nspec$ which represents the concentration of each species at each grid point at a given point of time. Some common partitioning strategies are discussed further, with reference to the 4-D array:

$A(1 : nx, 1 : ny, 1 : nz, 1 : nspec)$ - The 4-D data array

nx - number of elements in the first dimension (x)
ny - number of elements in the second dimension (y)
nz - number of elements in the third dimension (z)
$nspec$ - number of elements in the fourth dimension (number of species)

The general element of the array can be represented as $A(i, j, k, l)$. An x-vector can be represented as $A(:, j, k, l)$.

3.2.3.1 X-Y Partitioning.

The scheme involves partitioning the data into x-slices and y-slices, as shown in Figure 3.3. Mathematically, in a 4-D matrix, X-slice is the set of matrices $A(:, y, :, :)$ and the Y-slice is $A(x, :, :, :)$ where A is the full domain 4-D matrix (See Figure 3.3).

When using the implicit time-stepping for the 4-D data, the X-slice is suitable for X transport and the Z transport, while the Y-slice is suited well for Y transport and the Z transport. In the full integration, a reshuffling of X-slices and Y-slices is needed, at every step.

The disadvantage with X-Y partitioning for the implicit time-stepping case appears in the full transport scheme i.e. when combining X-Y-Z transport and chemistry. Because of the requirement of full vector(s) at every processor, a lot of data has to be exchanged between the processors. This adds to the communication time. If w_x, w_y are the widths of x-slices and y-slices respectively, the reshuffling for Y-transport computation just after performing an X-transport computation would imply transfer of chunks of data of size $w_x \cdot ny \cdot nz \cdot nspec$ from every processor that owns x-slice to the processor that owns a y-slice. Similarly the reshuffling for X-transport computation just after performing a Y-transport computation would imply transfer of data of size $w_y \cdot nx \cdot nz \cdot nspec$ from every processor that owns y-slice to the processor that owns an x-slice.

If an explicit time-stepping is used, then the data to be transferred between the processors is considerably less, with each processor owning the x-slice requiring the data from only the immediate neighbors in the domain. If W is the window of the explicit time-stepping used, then the reshuffling for Y-transport computation just after performing an X-transport computation would imply transfer of data of size $W \cdot ny \cdot nz \cdot nspec$ from the processors sharing boundary with the particular processor that owns an x-slice.

It should be clear in the further discussion that other partitioning schemes are more efficient than the X-Y partitioning both for the implicit and explicit time-stepping.

3.2.3.2 HV partitioning.

HV partitioning stands for horizontal vertical partitioning. The H slice is the sub-domain $A(:,:,z,:)$ of the whole domain matrix A, while V slice is the sub-domain $A(x_1 : x_2, y_1 : y_2, :, :)$

of the whole matrix A (See Figure 3.4).

When using the implicit methods, the H slices are appropriate for the X-Y transport while the V slices are appropriate for the z-transport and chemistry computations. However, once performing the full transport and chemistry computations the data would need to be shuffled between the processors that own H slices and the ones that own V slices. If w_V is the dimension of a V-slice, the reshuffling for Z-transport computation just after performing a horizontal-transport (X-Y) computation would imply transfer of data of size $w_V \cdot w_V \cdot nz \cdot nspec$ from every processor that owns a v-slice to the processor that owns an h-slice. Since there is not shuffling for horizontal transport, the dependency on number of x-slices and y-slices is removed. This makes the parallel application scalable to a higher number of processors.

This technique has been reported to be better for the implicit methods than the X-Y partitioning scheme [6].

3.2.3.3 Tiled Partitioning.

Tiled partitioning implies partitioning of data into x-y blocks. A tile is the sub-domain $A(x_1 : x_2, y_1 : y_2, :, :)$ of the whole domain matrix A (See Figure 3.5).

The tiled partitioning scheme alone is impractical for the implicit time-stepping methods scheme since the whole vector(s) would be needed at every processor. The reshuffling for Z-transport computation just after performing a horizontal-transport (X-Y) computation would imply transfer of data of size $w_x \cdot w_y \cdot nz \cdot nspec$ from every other processor to a particular processor that owns a tile.

However, when using a method with explicit time-stepping, tiled decomposition is more efficient. The scheme is appropriate because the communication needed at every stage is only with the immediate (non-diagonal) neighbors. The only data that needs to be communicated to the immediate neighbors is the data at the border (boundary) of the tiles. If W is the

window of the particular explicit time-stepping used, and w is the dimension of every tile, there would a transfer of data of size $W \cdot w \cdot nz \cdot nspec$ from every neighbor to a processor that owns a tile (See Figure 3.5).

3.3 Parallelization of Stem

For parallelizing an application one needs to identify the computationally intensive parts of the program and assess the time involved in communication that a particular scheme would imply. For AQMs of interest in the current work, the chemistry-related computations turn out to be computationally most expensive. In the serial version of the program, approximately 92% of the total time is spent towards chemistry-related computations. Only 7 % of the total time is spent towards transport, out of which around 4% of time is spent towards vertical transport and 3% towards horizontal transport.

Definitely, distributing chemistry computations effectively is of central importance to any scheme adopted for the parallelization of STEM. The nature of chemistry computations is such that the calculations at every grid point (x,y) are completely independent of the states of neighboring grid points. This implies that the domain can be divided into any desired finer granularity, as far as the chemistry-related computations are concerned. However, reducing the sub-domain to a very small block size would significantly increase the communication time for rest of the computations. (discussed further)

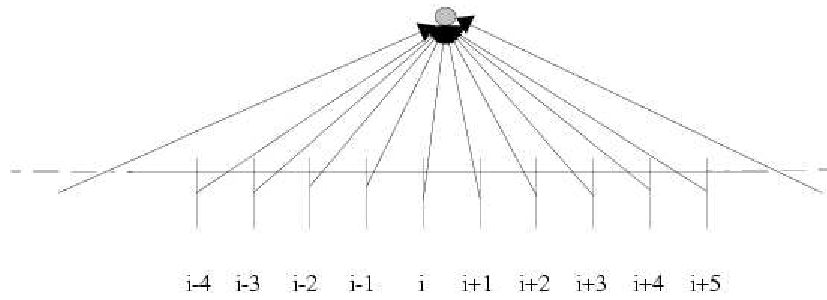
Although it is desirable to divide the domain into smallest number of blocks, an appropriate strategy needs to be adopted to ensure reasonable communication time for rest of the calculations.

The rest of the computations (other than chemistry) are that of transport, both horizontal and vertical. The vertical computations are similar to the chemistry-related computations

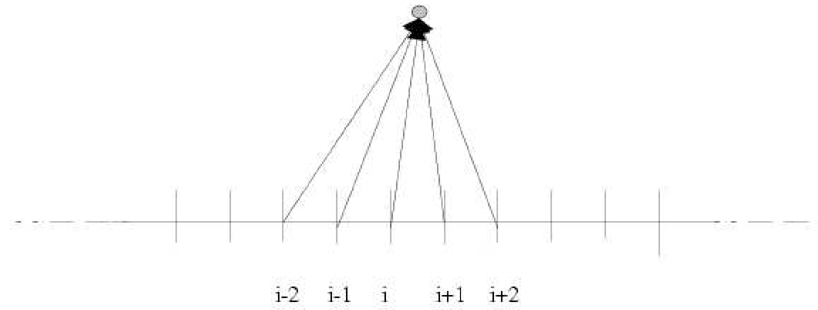
in the sense that every (x,y) grid point can be treated independent of neighboring points. However, the horizontal transport computations involve significant communication among the blocks.

The data dependency for the horizontal transport is such that every block needs only certain part of the data from the non-diagonal neighbor-blocks. If W is the window of the explicit time-stepping method being used, the data to be received from processors owning neighboring blocks is of size $(W \times nz \times nspec \times length_x)$ or $(W \times nz \times nspec \times length_y)$ where $length_x \times length_y$ are the dimensions of the block at the recipient processor. The data has to be exchanged at multiple stages at every iteration.

A common way to improve performance (as an attempt to get closer to the ideal case) is by letting computation and communication overlap. After sending the data to the communication line with a non-blocking call, the processor goes on performing the computations that don't require the data that it expects from the communication call. This strategy reduces the total time noticeably.

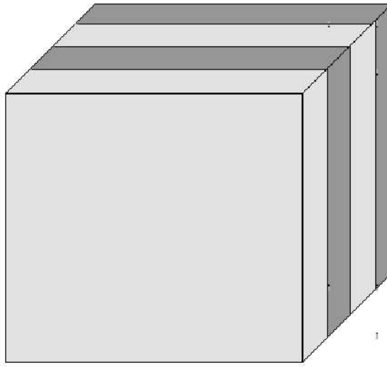


(a) Implicit

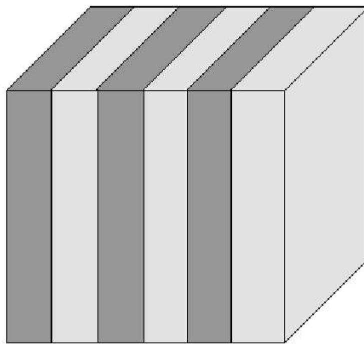


(b) Explicit

Figure 3.2. The domain of dependency (stencil) for 1-D spatial discretizations of the transport equation using Implicit and Explicit time-stepping schemes

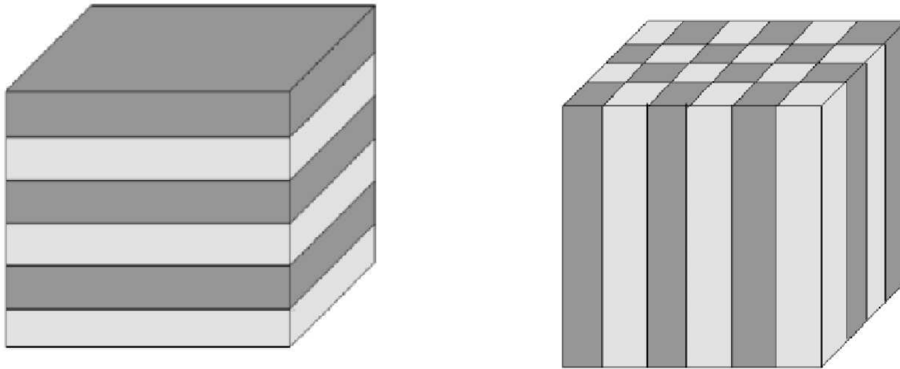


(a) x-slices



(b) y-slices

Figure 3.3. X and Y -slices in X-Y partitioning scheme



(a) H-slices (b) V-slices

Figure 3.4. H and V -slices in H-V partitioning scheme

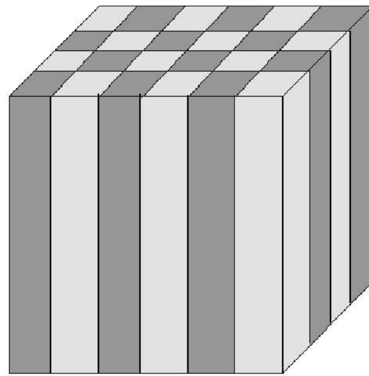


Figure 3.5. Tiled Partitioning

CHAPTER 4

IMPLEMENTATION

4.1 Environment and Tools

FORTRAN was used as the programming language for the STEM. The legacy code and the older implementations use f77 standards while all the newer code (except certain modifications to the older code) has been written using f90 standards. Apart from G95, the program has been ported on different systems, with absoft, ibm-xlf compilers.

Most of the performance tests were done on

- anantham - A 200+ processor AMD cluster at Virginia Tech. The machines run 64 bit Linux. The architecture in use is SMP.
- The terascale computing facility [13] - A cluster built with 1100 Apple Xserve G5 machines. They machines are connected with 4 SilverStorm Technologies 9120 InfiniBand core switches (4X InfiniBand, 10 Gbps bidirectional port speed 12.25 teraflops)

Both the clusters uses the PBS batch management system [14] for interfacing with the users that submit the jobs. PBS stands for Portable Batch System, a flexible batch queuing system developed for NASA in the early to mid-1990s.

4.2 Libraries

4.2.1 MPI.

For communication between the processors (task distribution-assimilation) we use the message passing paradigm. The software library used for this purpose was MPI (Message Passing Interface) MPI [15] has been extremely popular in the high-performance community, and is open source library.

The message passing interface is an implementation of the message passing paradigm for the communication between processors. The control of every processor is provided to the programmer, abstracting the low-level details of the communication (e.g. sockets).

The communication can be of the following types:

1. point to point sends and receives
2. broadcasts
3. distribution and collection related calls

We have used the mpi-ch implementation of the MPI which can easily be configured for many compilers. The development of this project was done on a unix workstation using g95 compiler.

4.2.2 NETCDF.

NetCDF (network Common Data Form) is an interface for array-oriented data access and a library that provides an implementation of the interface. The NetCDF library also defines a machine-independent format for representing scientific data. [16]

The NetCDF software was developed by Glenn Davis, Russ Rew, Steve Emmerson, John Caron, Harvey Davies, and Ed Hartnett at the Unidata Program Center in Boulder, Colorado, with contributions from many other NetCDF users.

NetCDF provides easy self-describing and portable data file; the file data can be easily accessed and interfaced with programs. The data from a NetCDF file can be modified, copied and archived as well.

ioapi is the library that interfaces with the NetCDF files. The I/O routines in STEM call the ioapi routines within.

In future, the HDF-5 standards are planned to be used. HDF-5 is a newer Hierarchical Data Format product, that supports larger files and more number of objects per file. The HDF-5 data model is simpler, and more comprehensive with only two basis structures (multidimensional arrays and group structure). It extends support to parallel I/O and multi-threading as well.[17]

4.2.3 LAPACK.

LAPACK is written in Fortran77 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems.It handles dense and banded matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision. [19]

4.2.4 BLAS.

The BLAS (Basic Linear Algebra Subprograms) routines provide standard building blocks for performing basic vector and matrix operations. [18]

4.3 Modular Design of STEM

The application was divided into the following modules -

1. Driver
2. Computation
3. Communication
4. Performance Analyzer
5. Debug

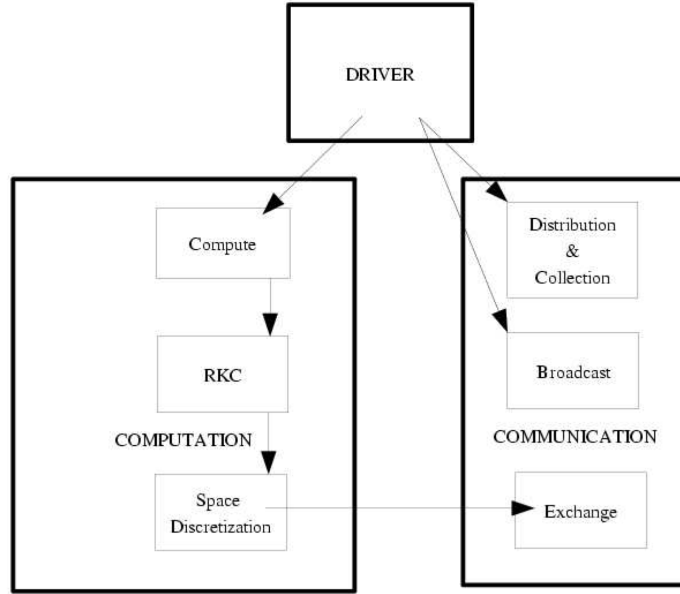


Figure 4.1. Modules in the program

Driver provides the entry to the program. Driver reads in the data files using the routines exported by IO modules (that use netCDF within) and calls the high-level routines for processing and presenting data.

Computation module contains the implementation of Runge Kutta Chebyshev method along with the derivative calculation method. It uses the communication calls form communication module to exchange data between the processors.

Communication module implements the functions required for all data communication between the processors. The communication routines are broadly of the following types -

1. Broadcast - The data that is needed by all the processors. These include the input parameters for various methods/techniques.

2. Distribution - The data that is read by the root and is sent to all the processors.

3. Exchange - The data that is exchanged among the processors.

4. Collection - The aggregation of the data at root.

Performance analyzer does the profiling of the program. It has been used for analyzing the execution times of various portions of the program and comparing the computation vs computation times.

Debug module implements general debug and trace routines. They proved to be very useful while the debugging and analysis of the parallel program.

4.4 Topology

For ease of referencing, a coordinate system is setup as shown in Figure 4.2. Ranks of the processors are mapped one-to-one onto the (x,y) coordinates in the system. The coordinate system helps, for example, in finding out the ranks of the neighbors a processor needs to communicate with.

The size of each tile depends on the total number of processors bearing the following relation:

$$length \cdot width = (nx \cdot ny)/np$$

where,

nx = total length of the matrix (size of the x-dimension)

ny = total width of the matrix (size of the y-dimension)

np = number of processors

The recommended values for width and length of the blocks are

$$length = width = \sqrt{(nx \cdot ny)/np}$$

It can be proved analytically that the above relation would result in optimal execution time, however, the user has been given the freedom to set the desired values of width and length for the block.

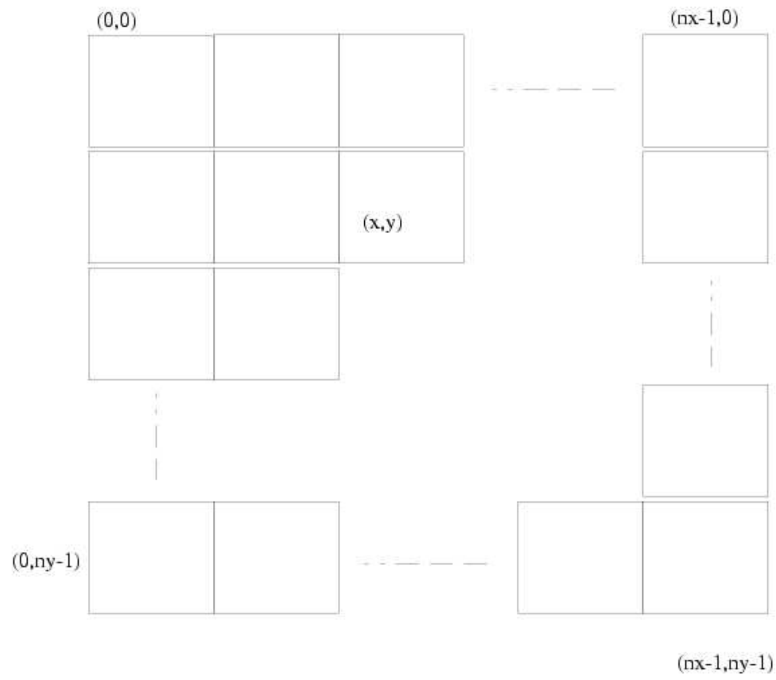


Figure 4.2. The coordinate system for bookkeeping tiles- Every processor is mapped to a point in the system

4.5 Code Flow

The typical run of the program goes through the following steps :

1. Initial Setup
2. Read Data Files
3. Broadcast
4. Distribution
5. Computation
6. Exchange
7. Collection/Submission
8. Cleanup and Exit

4.5.1 Initial Setup.

The Initial Setup involves reading environment variables (e.g. directory names to be used for I/O) and setting up initial parameters for the various computation modules used in the program.

4.5.2 Read Data Files.

Data files are read using the NetCDF routines. The data read is stored in large arrays that in turn, belong to respective fortran modules.

4.5.3 Broadcast.

Some of the input parameters that are read from the NetCDF input files, are needed at all processors. Such parameters are bundled together, and broadcast-ed using the `MPI_Bcast` routine.

4.5.4 Distribution.

4.5.4.1 Distribution of domain data.

The 4-D matrix on which the computation has to be performed, is partitioned in an a tiled decomposition manner and distributed to all the processors. This means that every processor would own the a x-y tiled partition of the domain. If the total domain is represented as $A(1 : nx, 1 : ny, 1 : nz, 1 : nspec)$, then every processor would have the tile $W = A(x_1 : x_2, y_1 : y_2, 1 : nz, 1 : nspec)$ after the tiled decomposition of the domain. Here, x_1, x_2, y_1, y_2 are bounding indices of the tile in the (full) domain.

The other 4-D matrices that are distributed to all the processors in a similar fashion are `sl1, kh` and `sp1`. There are various 2-D and 3-D matrices that are distributed among all the processors in a tiled fashion.

The implementation of tiled partitioning scheme

If the desired width of the block is a factor of total width of the matrix, the total number of blocks along the width are simply

`nX/wax`, where

```
nx = width of the matrix

wax = width of the block at each processor
```

If however, `wax` is not a factor of `nx`, the block of width same as the remainder `nx%wax` is created.

```
if (mod(nX,wax).gt.0) then
```

```
nax = nX/wax + 1

else

nax = nX/wax

end if
```

Similarly, if the desired length of the block is a factor of total length of the matrix, the total number of blocks along the length are simply:

`nY/way`,

where

```
ny = length of the matrix

way = length of the block at each processor
```

If `way` is not a factor of `ny`, the block of width same as the remainder `ny%way` is created.

```
if (mod(nY,way).gt.0) then

nay = ny/way + 1

else

nay = ny/way

end if
```

The root processor hence sends the sub-domains of the appropriate size (determined as above) to respective processors.

The coordinate system

A coordinate system is created for ease of programming and reusability. This is done with the `MPI_Cart_create` call provided by MPI (Figure 4.2).

```
ndims = 2

dim_size(1) = nax

dim_size(2) = nay

periods (1) = .false.

periods (2) = .false.

reorder = .true.

call MPI_Cart_create(com_world,ndims,dim_size, &

periods, reorder, comm_worker,ierr)

call MPI_Cart_coords(comm_worker,myid,2,coords,ierr)

x=coords(1)

y=coords(2)

if (x.eq.(nax-1).and.mod(nX,wax).gt.0) then

ax = mod(nX,wax)

else

ax = wax

end if
```

```
if (y.eq.(nay-1).and.mod(nY,way).gt.0) then

ay = mod(nY,way)

else

ay = way

end if
```

Every processor can know if it possesses a block that touches the domain boundary with just the id of the processor. A processors's id corresponds to an x-y position in the coordinate system. Due to the tiled decomposition of arrays, a processor on coordinate bounds would possess a block(subdomain) that lies on the boundary of the whole domain as well. Hence a processor just needs to check if its on the boundary of the coordinate system to find out if it has a block that lies on the boundary of the domain.

This also implies that a processor can easily determine the id of the processor it needs to send the data to, with a call to `MPI_Cart_rank` passing the x-y position (in the coordinate system) as arguments. For example, if root needs to send data to all processors owning blocks lying on first x-boundary of the domain, it just needs to pass 0 as the y-coordinate to the function `MPI_Cart_rank` and vary the x-coordinate from 0 to maximum value of x in the coordinate system.

4.5.4.2 Distribution of boundary data.

The root doesn't need to send the boundary data to all the processors. The boundary data are sent only to the processors that would be using the the data in the computations, i.e. the processors that are on the boundaries of the topology in use (Figure 4.13).

The root finds out what processors own the blocks(subdomains) lying on the domain boundary by varying x and y coordinates as arguments to the MPI function `MPI_Cart_rank` :

```
coords(1) = xi !  varied from 0 to nax -1

coords(2) = 0

call MPI_Cart_rank(comm,coords,receiver_rank,ierr)

...

coords(1) = xi !  varied from 0 to nax -1

coords(2) = nay-1

call MPI_Cart_rank(comm,coords,receiver_rank,ierr)

...

coords(1) = 0

coords(2) = yi !  varied from 0 to nay-1

call MPI_Cart_rank(comm,coords,receiver_rank,ierr)

...

coords(1) = nax-1

coords(2) = yi !  varied from 0 to nay-1

call MPI_Cart_rank(comm,coords,receiver_rank,ierr)
```

4.5.5 Computation.

The driver advances the system by one time-step by performing the following sequence of calls:

for the simulation time

{

call X-Y transport

call Z transport

call chemistry

call Z transport

call X-Y transport

}

All the processors perform computations on the data received from the root process. The RKC function needs to evaluate derivatives on the received block, s times at every iteration.

The calculation of the derivative for horizontal transport at every point requires values from 2 neighboring points in every direction. In other words the window size for the explicit time-stepping is 2. (See Equations (2.7)(2.8) (2.9) (2.10)) For calculation of the derivative in the tiled partitioning that is being used here, the block that lies in the middle of the topology surrounded by neighboring blocks would require values of two points each for every point on the boundary that is shared with the neighboring block. In this manner, the whole block would need arrays of data points from every block that is its neighbor (See Figure 4.4).

Every processor hence needs an array of what are termed as ghost cells - the values of the concentration (4D block) from the previous iteration at the points that lie outside the

domain that the processor owns (See Figure 4.4). The ghost-cell exchange would be needed at multiple stages per iteration. There is no need of any exchange for chemistry computations and the z-transport.

However, the calculation of derivative at the boundary of the data needs the boundary data only. The block that lies at the boundary of the topology doesn't have any neighbor at the respective boundaries, hence requiring limited data exchange (See Figure 4.7).

For every processor to know whether the block it owns touches the boundary of the domain or not, the program maintains a vector of logical/boolean variables - t_N, t_E, t_W, t_S . The id of the processor is mapped with the vector of logicals (length=4) indicating whether the block touches the boundary.

For example, id 0 corresponds to $[t_N, t_E, t_W, t_S] = (1,0,1,0)$ - implies the block touches boundary in north and west direction (corresponding to the shaded area in fig 4.5. (The sense of north, east, west and south is only to represent the directions in the adopted coordinate system, and have no connection with the geographic directions)

The ghost cell data obtained from the respective neighboring processors is appended to the block at the recipient processor in a fashion depicted in Figure 4.9. This matrix is then passed on to the RKC function where the derivative calculation is performed. Since the derivative is not evaluated at the grid points obtained as ghost cells the program uses certain markers for updating only the grid points that are not part of the ghost array (See Figure 4.10). The calculation of derivative involves visiting every grid point sequentially. Hence, the derivative function visits the point `(x_start,y_start)` and goes until the point `(x_end,y_end)`. The enhanced block is of dimensions (`xn` × `yn` × `nz` × `nspec`), and the final matrix returned to the caller is of dimensions (`x_end-x_start+1`) × (`y_end-y_start+1`) × `nz` × `nspec`.

The final matrix that the computation function returns is obtained by removing the ghost

cells boundary from the enhanced matrix.

Other than the concentration matrix on which the derivative is computed, there is another data array - Air which is used in the calculation of derivative as well. Air is dependent on the 4-D concentration matrix, and is assigned the respective values at every call to transport function.

4.5.6 Exchange.

As mentioned before, the ghost cell exchange involves communication between only the immediate neighbors. The processors having the sub-domain along the boundary, don't need exchange along the boundary.

After every stage of computation, every processor needs to share some data with the neighboring processes to move on to the next stage (Figure 4.6). This data referred as the ghost cells (Figure 4.8) is exchanged using the MPI derived data types at every stage of the computation.

Also since the blocks that touch the boundary won't need to exchange any data in the direction they touch the boundary (Figure 4.13) the exchange function detects the location of the block in the topology and decides the blocks to exchange the data with.

In the design where root is involved in the computation as well, the root doesn't receive any data in the first exchange, since it has all the data it needs to perform the computation. However, to calculated the next stages of computation, it does need the data computed by the neighboring blocks.

The MPI implementation of the ghost cell exchange was done with

- A combination of `MPI_Isend` and `MPI_Recv` - The processor sends the ghost cell data to its neighbors in all the directions using a call to `MPI_Isend` and then waits as the

recipient processor for the ghost cell data it needs from its neighbors, using a call to `MPI_Recv`. It then checks the `MPI_Isend` to have finished using a call to `MPI_Wait`.

- MPI_Sendrecv - The sending and receiving take place hand in hand with the a single call to `MPI_Sendrecv`.

With both the approaches the execution times were about the same.

4.5.7 Collection/Submission.

One the computation is over, all the processors submit data to the root. The root gathers data from all the worker processors, and performs the file-read, distribution and computation repeatedly until the simulation has run for the requested number of hours.

4.5.8 Exit.

After collecting computed data from all the processors, the root process stops the timer, performs the rest of the cleanup activity (e.g. Memory deallocation related) and ensures a clean exit of the program.

The data types and functionality of the data collection are equivalents of those involved in distribution of data.

4.6 Data Types used in the code

Most of the data types in the program use the derived data types provided in MPI. The data communication doesn't involve packing/unpacking of data to be sent/received. Buffer copies have been minimized to the possible extent.

4.6.1 Data types for Broadcast.

In addition to the large data matrices partitioned and distributed by the root, there are a number of initial parameters needed on every processor to start the computation. Such data (integer, reals, and double precision) were bundled intro arrays and sent using `MPI_Bcast`, avoiding more than one call to `MPI_Bcast`.

4.6.2 Data types for Distribution.

The root processor is responsible for sending the large data matrices to every other processor. The 4-D data that every processor receives is of dimensions ($ax \times ay \times nz \times nspec$) where

ax, ay are the dimensions of the x-y block (Figure 4.3).

The data types needed for whole domain were defined for sending data as follows

```
1 call MPI_TYPE_VECTOR (ay,ax,nX,MPI_REAL,Block,ierr)
2 call MPI_TYPE_COMMIT (Block,ierr)

3 call MPI_TYPE_HVECTOR (nZ,1,sizeofDble*nX*nY,Block,Block2,ierr)
4 call MPI_TYPE_COMMIT(Block2,ierr)

5 call MPI_TYPE_HVECTOR (Nspec,1,sizeofDble*nX*nY*nZ,Block2,Block4,ierr)
6 call MPI_TYPE_COMMIT(Block4,ierr)
```

Here, for the type 'Block', in the call `MPI_TYPE_VECTOR` (at lines 1,2)

ay - count

ax - length

nx - stride

When the data is received at every processor, the data stride is ax instead of nx (Figure 4.3)

```
1 call MPI_TYPE_VECTOR (ay,ax,ax,MPI_DOUBLE_PRECISION,Block,ierr)
2 call MPI_TYPE_COMMIT (Block,ierr)

3 call MPI_TYPE_HVECTOR (nZ,1,ax*ay*sizeofDble,Block,Block2,ierr)
4 call MPI_TYPE_COMMIT(Block2,ierr)

5 call MPI_TYPE_HVECTOR (nospec,1,sizeofDble*ax*ay*nZ,Block2,Block4,ierr)
```

```
6 call MPI_TYPE_COMMIT(Block4,ierr)
```

Similarly, for 4-D data other than concentration (sl1, sp1)

```
1 call MPI_TYPE_HVECTOR (n_liquid,1,sizeofDble*ax*ay*nZ, &
Block2,Block4_nl,ierr)
2 call MPI_TYPE_COMMIT(Block4_nl,ierr)

3 call MPI_TYPE_HVECTOR (n_liquid,1,sizeofReal*ax*ay*nZ,&
r_Block2,r_Block4_nl,ierr)
4 call MPI_TYPE_COMMIT(r_Block4_nl,ierr)

5 call MPI_TYPE_HVECTOR (n_particle,1,sizeofDble*ax*ay*nZ,&
Block2,Block4_np,ierr)
6 call MPI_TYPE_COMMIT(Block4_np,ierr)

7 call MPI_TYPE_HVECTOR (n_particle,1,sizeofReal*ax*ay*nZ,&
r_Block2,r_Block4_np,ierr)
8 call MPI_TYPE_COMMIT(r_Block4_np,ierr)
```

The similar data structures are created for real (instead of double precision) data.

4.6.3 Data types for Exchange.

For exchanging Kh (a 3-D matrix) appropriate data types for ghost cells need to be defined.

```
1 call MPI_TYPE_VECTOR (nZ,ay,ay,MPI_DOUBLE_PRECISION,Blockyz,ierr)
2 call MPI_TYPE_COMMIT (Blockyz,ierr)

3 call MPI_TYPE_HVECTOR (width,1,sizeofDble*ay*nZ,Blockyz, &
GhostArrayX,ierr)
4 call MPI_TYPE_COMMIT(GhostArrayX,ierr)

5 call MPI_TYPE_VECTOR (nZ,ax,ax,MPI_DOUBLE_PRECISION,Blockxz,ierr)
6 call MPI_TYPE_COMMIT (Blockxz,ierr)

7 call MPI_TYPE_HVECTOR (width,1,sizeofDble*ax*nZ,Blockxz, &
GhostArrayY,ierr)
8 call MPI_TYPE_COMMIT(GhostArrayY,ierr)
```

The aim is to define an array of a variable width (= the width of the ghost cell depending on the algorithm) and of length corresponding to the block being computed (The RKC methods used in this work set $width \leq 2$) (See Figure 4.8).

Similar data structures are needed for the exchange of concentration as well (4-dimensional matrix)

```
1 call MPI_TYPE_VECTOR (nZ,ay,ay,MPI_DOUBLE_PRECISION,Blockyz,ierr)
2 call MPI_TYPE_COMMIT (Blockyz,ierr)
```

```
3 call MPI_TYPE_HVECTOR (width,1,sizeofDble*ay*nZ,Blockyz, &
Blockyzw,ierr)
4 call MPI_TYPE_COMMIT(Blockyzw,ierr)

5 call MPI_TYPE_HVECTOR(nspec,1,sizeofdble*ay*nz*width,Blockyzw, &
GhostArrayX,ierr)
6 call MPI_TYPE_COMMIT(GhostArrayX,ierr)

7 call MPI_TYPE_VECTOR (nZ,ax,ax,MPI_DOUBLE_PRECISION,Blockxz,ierr)
8 call MPI_TYPE_COMMIT (Blockxz,ierr)

9 call MPI_TYPE_HVECTOR (width,1,sizeofDble*ax*nZ,Blockxz, &
Blockxzw,ierr)
10 call MPI_TYPE_COMMIT(Blockxzw,ierr)

11 call MPI_TYPE_HVECTOR(nspec,1,sizeofdble*ax*nz*width,Blockxzw, &
GhostArrayY,ierr)
12 call MPI_TYPE_COMMIT(GhostArrayY,ierr)
```

The boundary values are distributed by the root processor and the respective data types bear similarity with the ghost array (they can be viewed as the ghost array of width 1)

```
1 call MPI_TYPE_VECTOR (nZ,ay,ay,MPI_DOUBLE_PRECISION,Blockyz,ierr)
2 call MPI_TYPE_COMMIT (Blockyz,ierr)
```

```
3 call MPI_TYPE_HVECTOR (Nspec,1,sizeofDble*ay*nZ,Blockyz, &
Block_bdryX, ierr)
4 call MPI_TYPE_COMMIT(Block_bdryX,ierr)

5 call MPI_TYPE_VECTOR (nZ,ax,ax,MPI_DOUBLE_PRECISION,Blockxz,ierr)
6 call MPI_TYPE_COMMIT (Blockxz,ierr)

7 call MPI_TYPE_HVECTOR (Nspec,1,sizeofDble*ax*nZ,Blockxz, &
Block_bdryY, ierr)
8 call MPI_TYPE_COMMIT(Block_bdryY,ierr)
```

The data structures required for collecting the data (at root) are same as that of distribution.

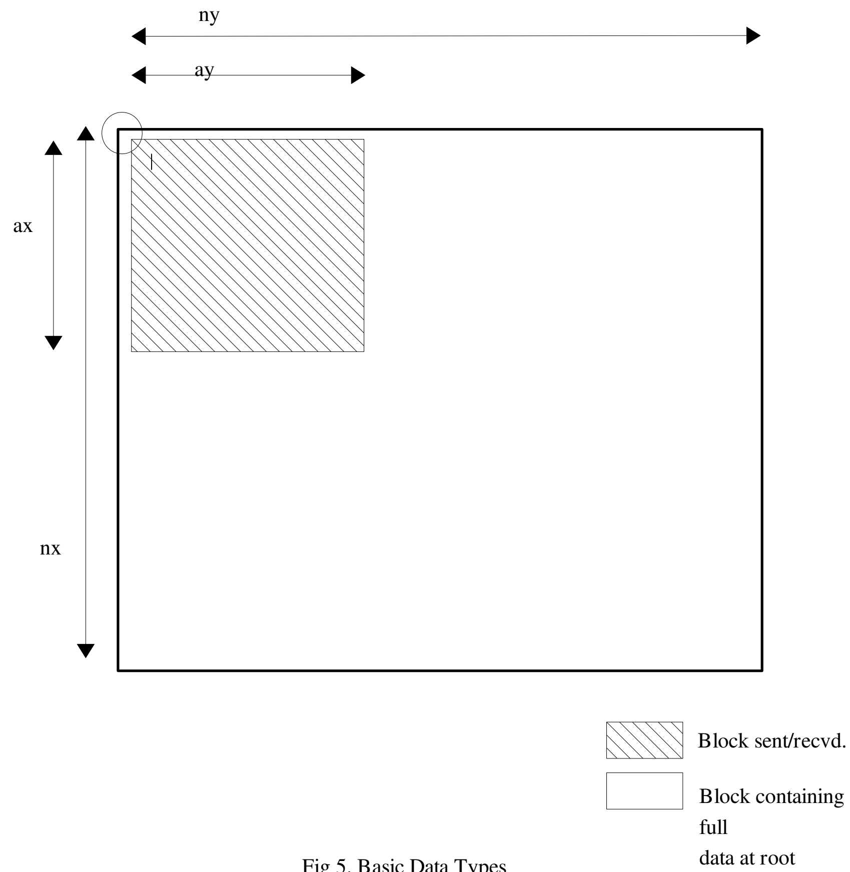


Figure 4.3. The dimensions of the basic datatypes used in distribution

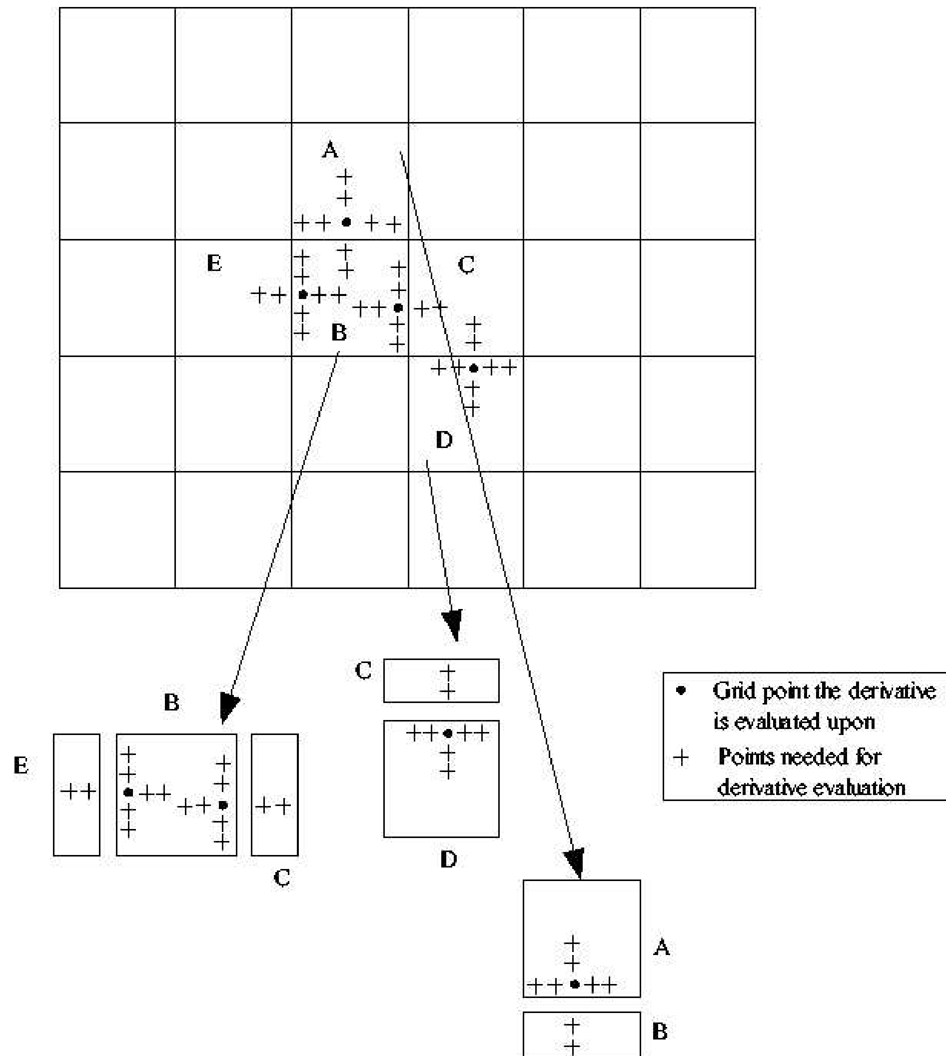


Figure 4.4. The need of ghost cells for the computation at every processor

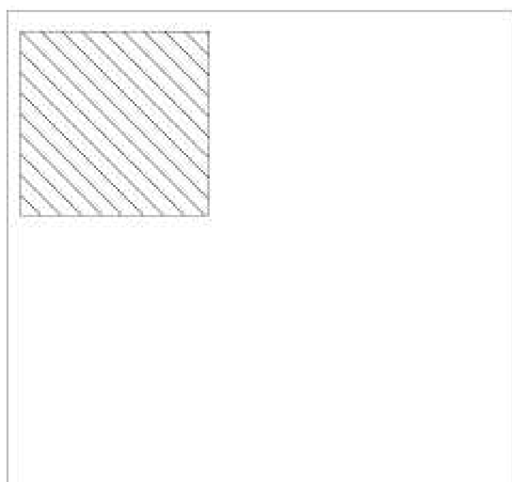


Figure 4.5. The block at root is the north-west block in program's terminology

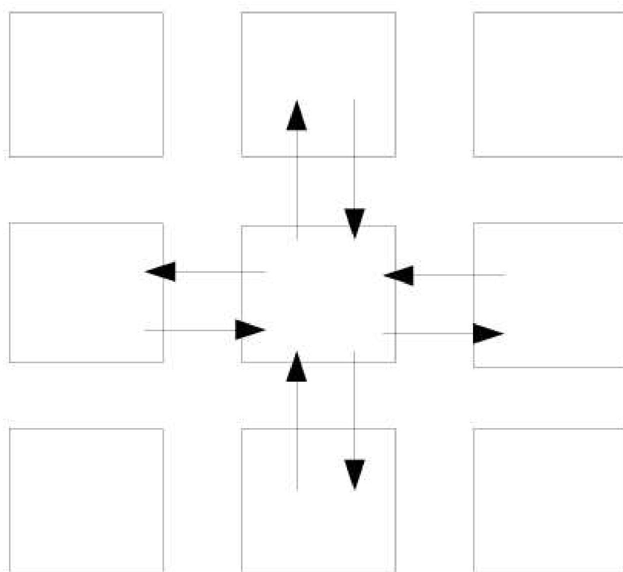


Figure 4.6. Exchange of data (ghost cells) at every stage

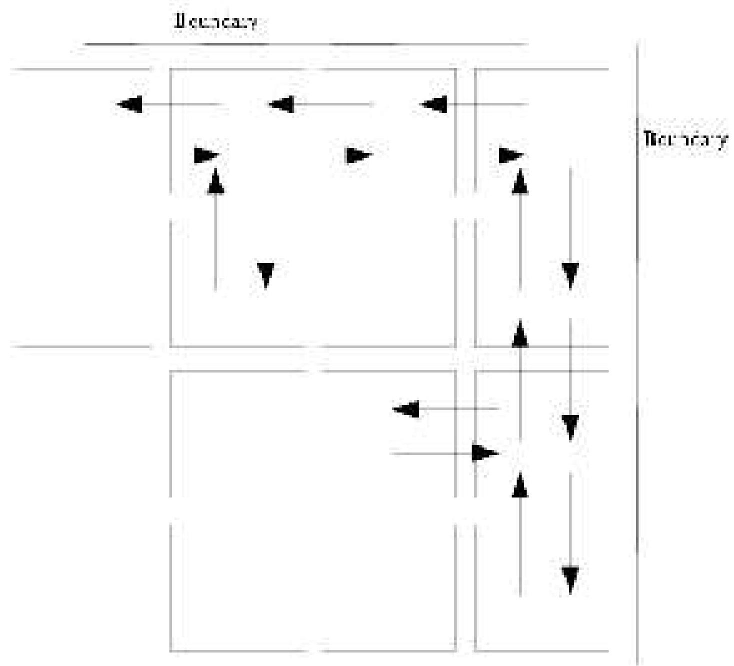


Figure 4.7. Blocks along boundary need limited exchange

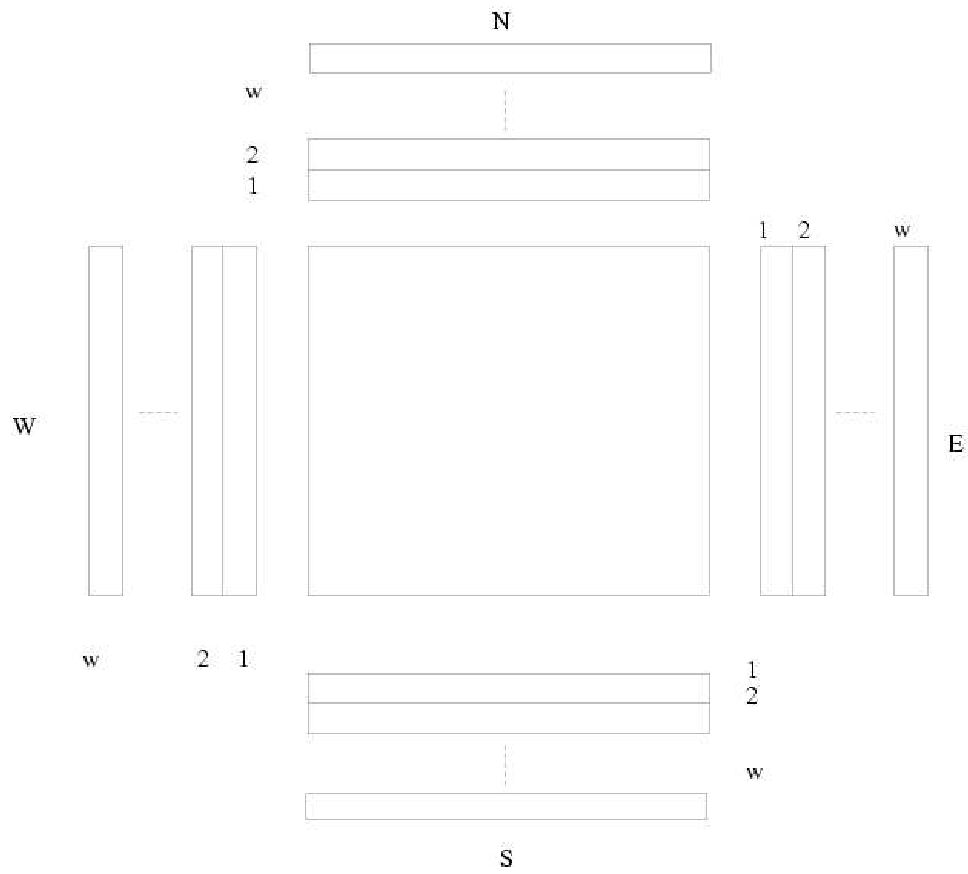


Figure 4.8. Ghost Cells are appended to the matrix at every block

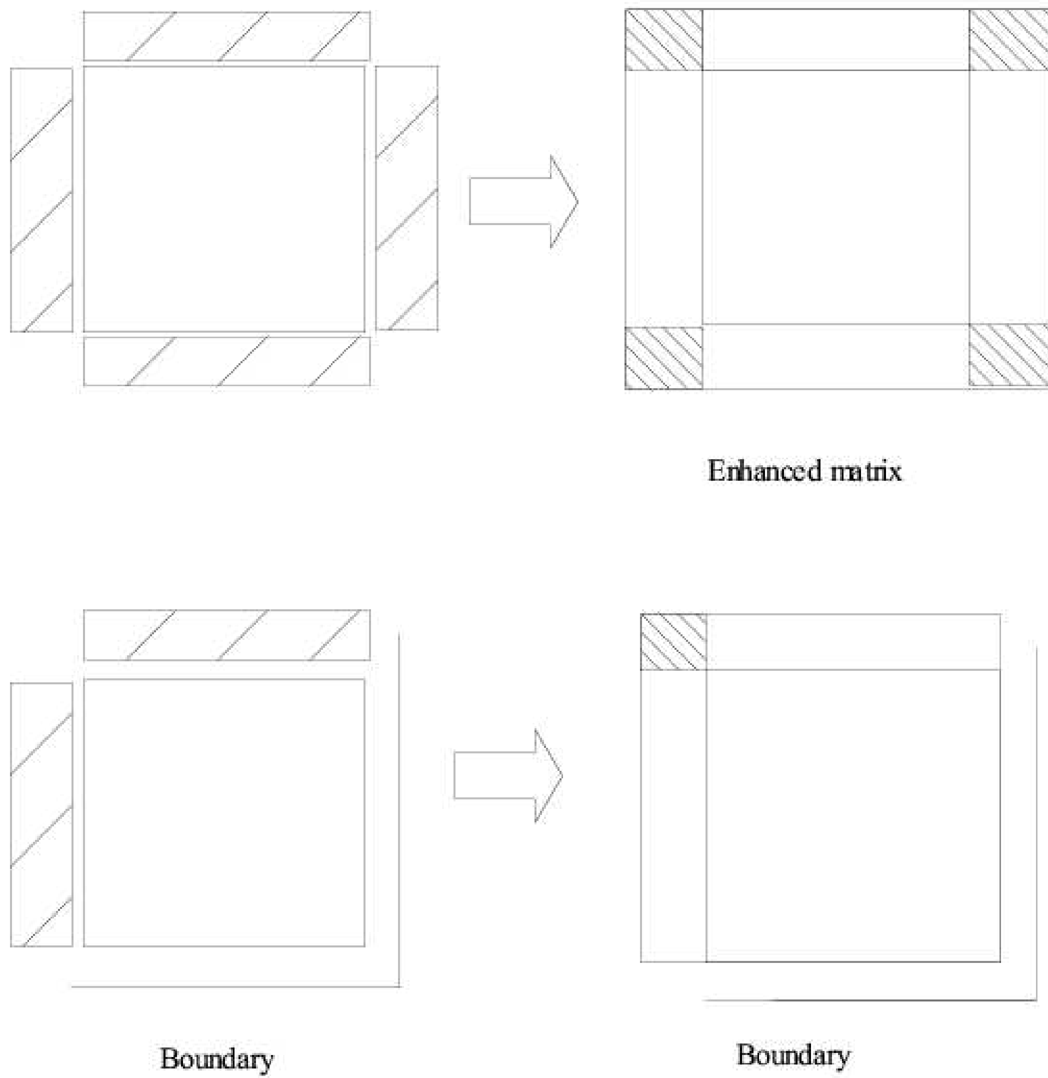
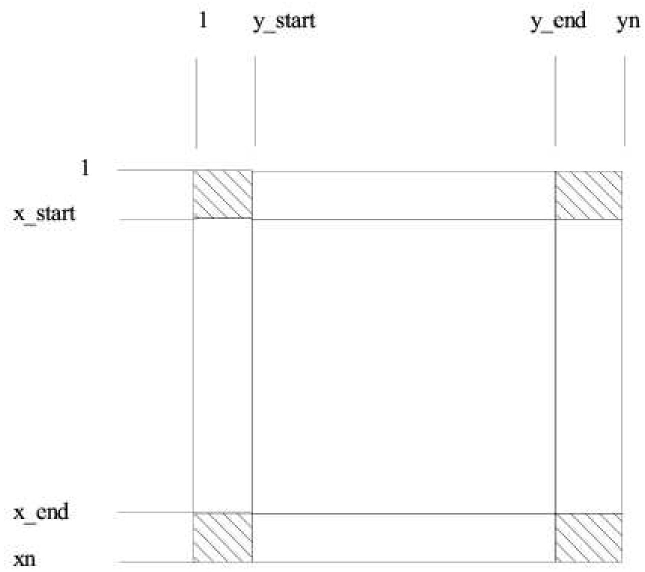
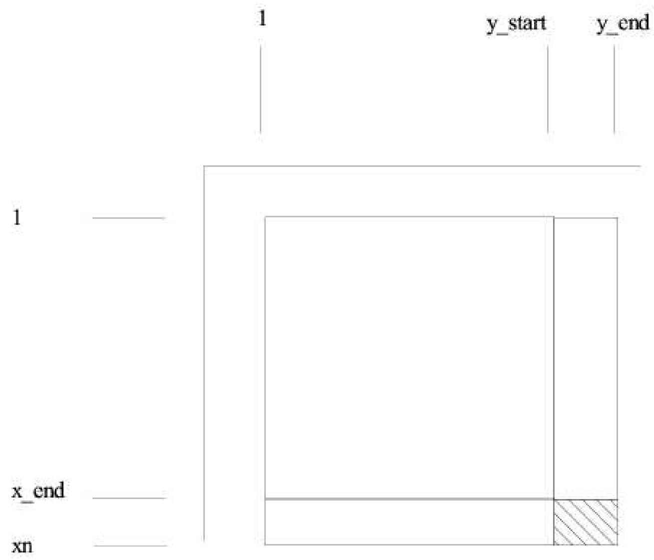


Figure 4.9. Enhanced matrix after appending the exchanged data from the neighbors



(a) markers for non-boundary block



(b) markers for boundary block

Figure 4.10. Markers used in the program for calculation of derivative on the enhanced matrix after appending ghost cells

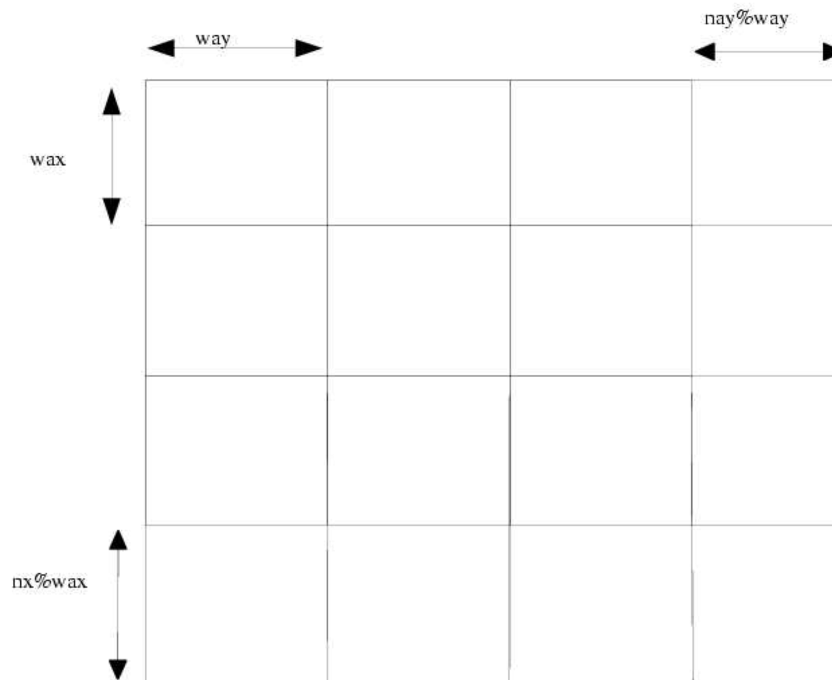


Figure 4.11. Assignment of block sizes

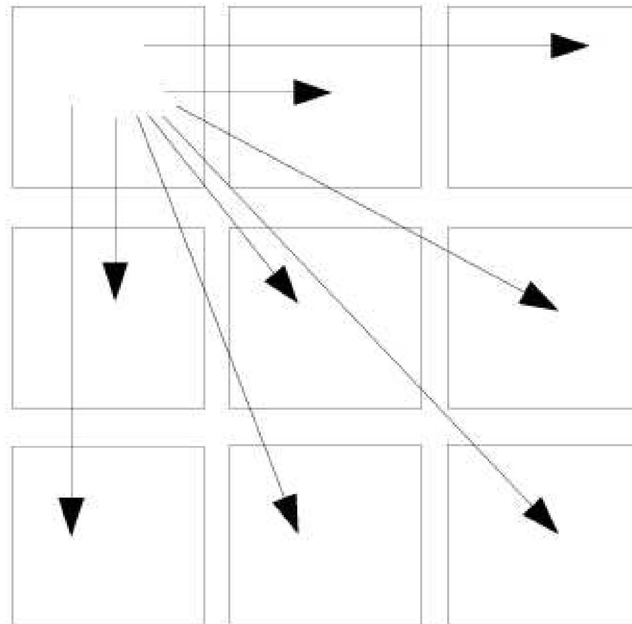


Figure 4.12. Distribution of input data (initial concentrations, wind fields and diffusion coefficients) to processors

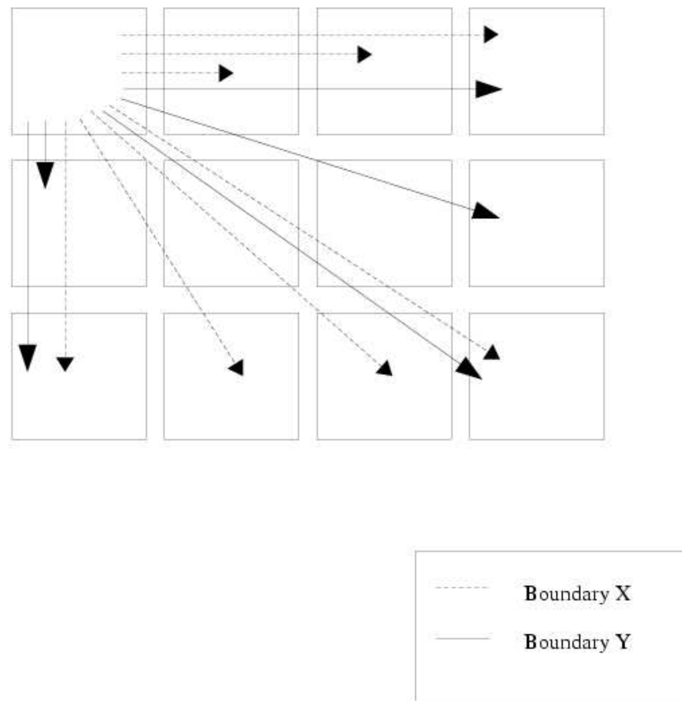


Figure 4.13. Distribution of boundary data to respective processors

CHAPTER 5

RESULTS AND DISCUSSIONS

5.1 Test Problem

The test problem used for the experiments run in the project was the meteorological data set for Asia. The data spanned over 90 $\times$ 60 horizontal grid points, while the altitude was over 18 vertical grid points. The number of species the simulation was run for was 106. In this manner, the 4-D matrix computed in the simulation was of size ($90\times60\times18\times106$). The data-types used were FORTRAN `double precision`, making the total size of the data to be 8.24×10^7 (bytes) or 82 MB (approximately). (See Figure 5.1).

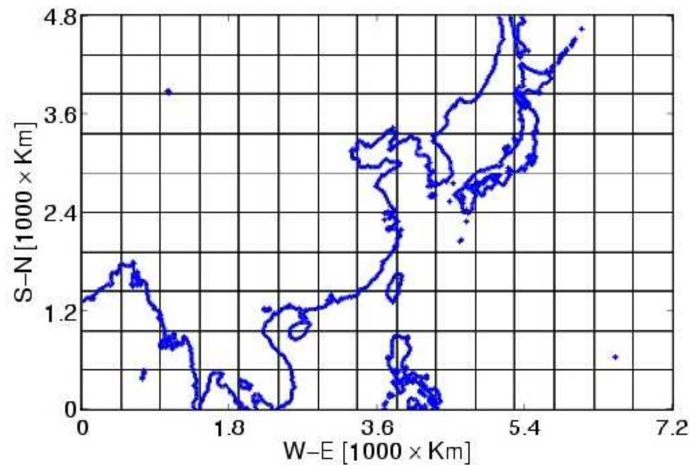
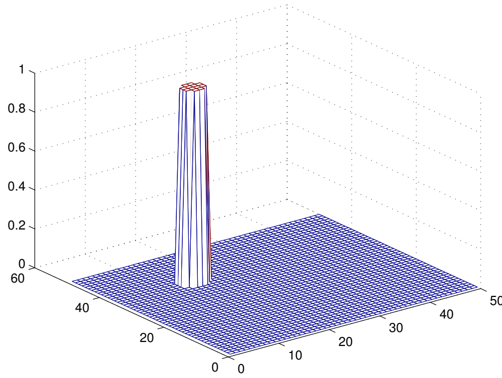


Figure 5.1. The grid-map of Asia - the region where the data were collected from

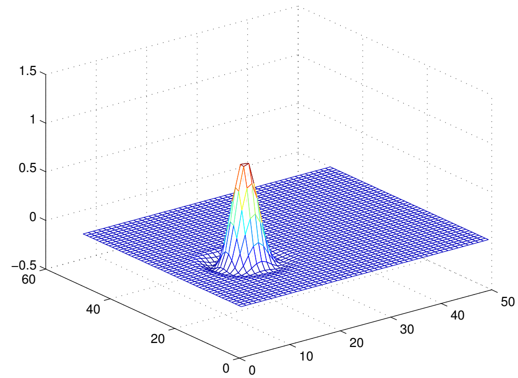
5.2 Preliminary Tests for RKC Time Integration

The integration of advection diffusion equations was tested with simpler experiments - Starting with a cylinder as the 2-D concentration matrix of size ($nX\times nY$) the solution matrix becomes closer to a bell-shaped curve under diffusion. If the wind field is circular, the

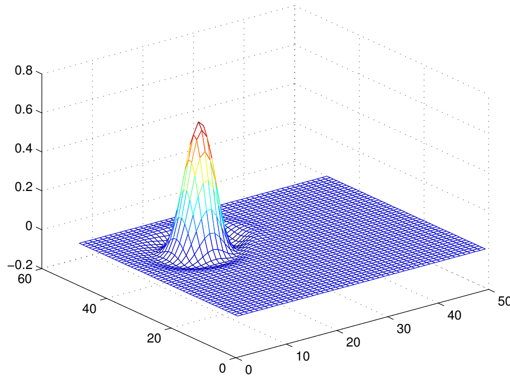
cylinder revolves around the axis of the circular wind field (Figure 5.2).



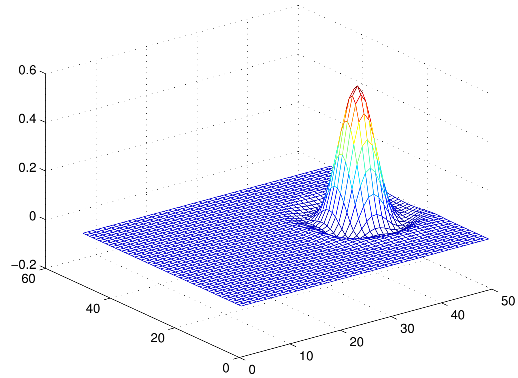
(a) Initial Concentration



(b) Solution after 9 h



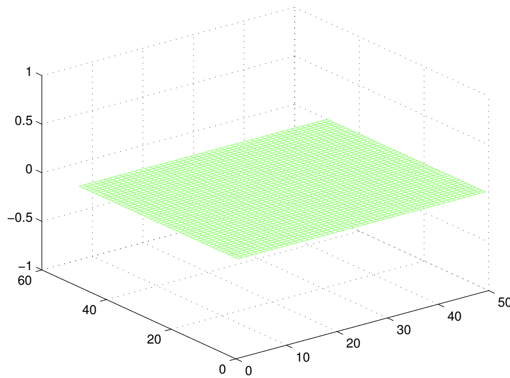
(c) Solution after 90h



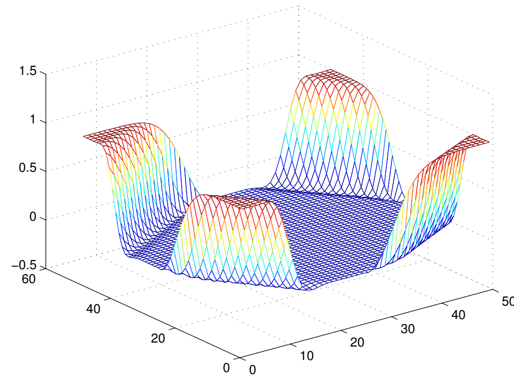
(d) Solution after 7 days 2h

Figure 5.2. The solution of advection diffusion equations at various times

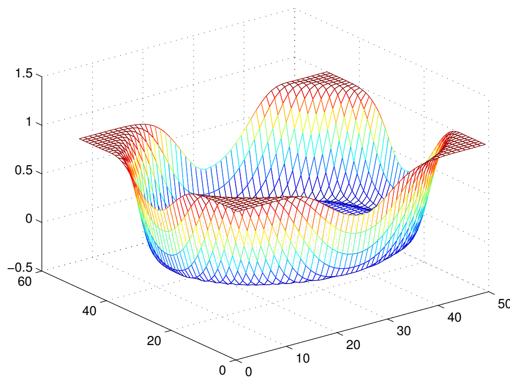
The other simple test experiment consists of setting up the boundary values to non-zero values while setting the concentration matrix to be zero. Eventually, the solution becomes closer to a dish/bowl in shape.



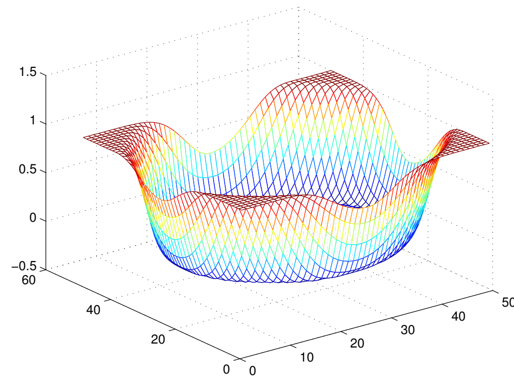
(a) Initial Concentration (boundaries=1)



(b) Solution after 9 h



(c) Solution after 90h



(d) Solution after 7 days 2h

Figure 5.3. The solution of advection diffusion equations at various times for numerical boundary conditions

5.3 Results

The results presented were obtained from the runs on parallel machines- anantham and System-X.

For profiling the application(s), the MPI call `PMPI_WTIME()` was used for the parallel machines while `cpu_time()` was used on the serial machine. The runs of the serial programs on both the machines reported higher running time of the explicit method.

5.3.1 The performance metrics.

The following metrics were recorded for the runs of the parallel program on upto 70 processors.

1. Execution time
2. Communication time
3. Computation time
4. Time spent in transport
5. Time spent in chemistry

Table 5.1. Time spent on anantham (seconds) for 1h simulation time

np	explicit ($s = 5$)	explicit ($s = 3$)	explicit ($s = 2$)	implicit
1	4829	4771	4730	4712
4	3458	3346	3231	1501
8	877	811	764	592
16	470	428	401	290
36	244	235	199	135
54	180	159	153	97
70	153	137	130	70

The plots of total time spent in the simulation are shown in Figure 5.6 and Figure 5.7.

Speedup is defined as wall-clock time of serial execution divided by wall-clock time of parallel execution.

$$Speedup = T_{serial}/T_{parallel}$$

Efficiency is defined as the ratio of speedup and the number of processors. This gives an idea of how well the application scaled/s to the number of processors.

$$Efficiency = Speedup/(Number\ of\ processors)$$

The speedup plot is shown in Figure 5.8. The break-up of time spent in chemistry and transport is shown in Figure 5.12. It is evident that the chemistry scales really well on the processors, while transport doesn't scale as well after a certain threshold (due to the nature of data decomposition we use).

The efficiency plots are shown in Figure 5.10 and 5.11.

It is observed that for the explicit method running on smaller number of processors, the communication is a lot more intensive. On profiling it was found that most of the time is

Table 5.2. Time spent on system-X (seconds) for 1h simulation time

np	explicit ($s = 5$)	explicit ($s = 3$)	explicit ($s = 2$)	implicit
1	1593	1542	1530	1298
8	255	230	227	237
16	129	135	119	127
30	77	73	68	83
36	67	62	56	71
54	50	46	44	55
70	39	37	34	47
78	35	30	28	41
90	32	29	28	40
100	33	28	27	42

spent in waiting, especially on smaller number of processors.As the number of processors increased the waits were lesser.

5.3.2 Predicted Execution Times vs the Observed Execution Times.

A simple model for predicting the execution times for the algorithm can be formulated. The model assumes that for transferring m bytes data from any processor A to any other processor B would be given by a simple formula:

$$T_m = T_s + T_b \cdot m$$

Here, T_m is the total time taken in sending the data of size m while T_s and T_b are machine constants. With a few experiments on system-x the values of T_s and T_b were of order 10^{-4} and 10^{-8} respectively.

As for the computation time(s), a fair assumption would be that the execution time varies linearly with the number of elements in the 4-D array. The more the number of elements

on which the spatial derivative is calculated, more should be the total time taken towards computation. With these assumptions, the predictions for distribution, computation and exchange are presented in Figure 5.4 and Figure 5.5 along with the observed respective execution times on system-X.

The time taken for distribution of a 4-D array was predicted with the following formula :

$$T_{distr} = (np - 1)(W_x \cdot W_y \cdot nz \cdot nspec)T_m$$

The time taken for computation of derivative on the 4-D array at every processor was predicted using the following formula:

$$T_C = (W_x \cdot W_y \cdot nz)T_{C1}$$

Here, T_{C1} is the time taken for evaluation of derivative at one grid point (The computation uses fortran vector multiplications, with each vector of size *nspec* i.e. the number of species).

It was found that the simple model used here, predicts the times for computation and distribution quite well. However, the prediction of time(s) spent in the exchange-communication is different from the observed execution times. The reason is that the model is too simple to predict the load imbalance that might cause the processors not to finish exactly after the same intervals of time after performing the same amount of calculation. The model assumes that all processors would execute the same amount of code in equal times. However, the reality is far away from that. In fact this discrepancy would be even more on anantham where the load imbalance was found to be higher.

5.3.3 Analysis of the results.

The primary reason for most of the transport-time spent towards waits is the load imbalance

among processors due to chemistry. If there is a significant load imbalance among the processors, the execution time for same amount code varies significantly. Since the RKC methods in use here, synchronize at multiple stages per iteration, this causes significant increase in the communication time through long waiting times. This affects the performance of the application on anantham more than it does on System-X. On anantham, efficiency reported by the explicit method is worse than the implicit method (See Figure 5.8).

Also, this effect is observed to be more prominent in smaller number of processors since the computation per processor is high for small number of processors. This results in a proportional difference between computation times for the processors. Higher the difference in computation times, higher are the waiting times at every exchange and hence the total communication time.

Apart from the load imbalance, the degradation in performance with increasing number of processors is due to lesser computation (with respect to communication) per processor with increasing number of processors.

The speedup plot obtained from System-X runs is shown in Figure 5.9. On System-X the execution times spent towards transport are reduced significantly. In the break up of execution times shown for implicit and explicit implementations in Figure 5.12, it is clear that the communication time towards transport reduces significantly with increasing number of processors in explicit implementation while it doesn't so in the implicit implementation.

The explicit technique is thus found to exhibit better parallelization efficiency on the System-X. It can be seen that the performance of the implicit starts to decline after 80 processors, unlike the explicit method.

Another important observation is that the time spent in distributing the data at root becomes more significant towards the total time as the number of processors is increased. This is

because as the number of processors is increased, the computation that has to be performed at every processor as well as the amount of data to be exchanged at every processor decreases. The root, on the other hand, has to distribute same amount of total data to more processors as the number of processors is increased (although with lesser amount of data to be received at every processor). The total time spent on distribution therefore, increases with increasing number of processors. This definitely, makes a strong case for moving towards a parallel I/O implementation (See Figure 5.12).

5.3.3.1 Effect of parameter s on execution time.

There are s stages at every iteration, which imply exchanging ghost cells s times with the neighboring blocks at every iteration. Every time a ghost cell exchange takes place, the processors have to be synchronized. Therefore, a lower value of s is more desirable.

If s can be set to a low value without compromising with the accuracy too much then the communication time can be reduced significantly (the compatibility of stability regions shown in Figure 2.2 is what is sought in the selection of s). In the experiments run on system-X, the time spent towards transport is significantly less for lower values of s. This is certainly more desirable in the case when there is significant load imbalance among the processors - less exchange among processors reduces time spent in transport noticeably.

5.3.4 Overlapping communication time and computation time.

To make the communication and computation time overlap, the general idea is perform the calculations that don't need the data being waited for in a `MPI_Recv` operation.

In the current implementation, this was done by calling non-blocking functions for block exchange, and then going ahead with the computations that didn't involve the data that was to be exchanged. After completing such computations the exchange was finished and the computations for the rest of the data followed.

In our case however, the introduction of such a scheme brings in certain book-keeping operations and causes multiple runs over the large array. No significant improvements over the old scheme are observed, as a result. The comparison of the overlapped communication-computation and the previous implementation is presented in Figure 5.13.

5.4 Software Configuration Management

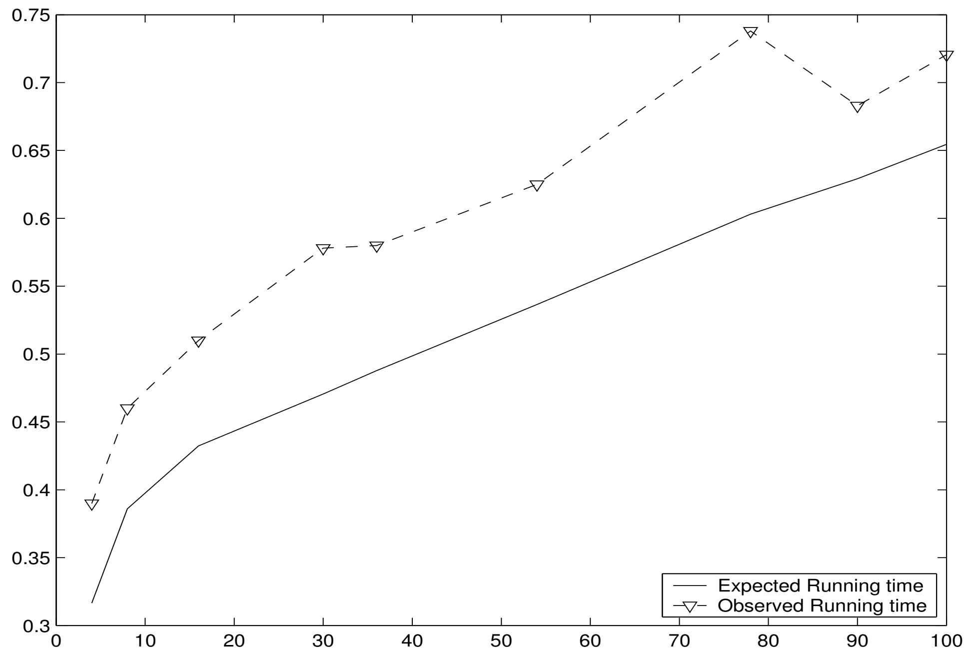
The following rules were followed for the configuration management.

1. All the independent and testable functions/modules were made to be CM entities.
2. Every version of the file listed the test-cases it has passed through
3. Every file contained the last updated / last modified tag
4. Makefiles were treated as CM entities, and contained comments related to the modifications/additions with respect to the particular environment.

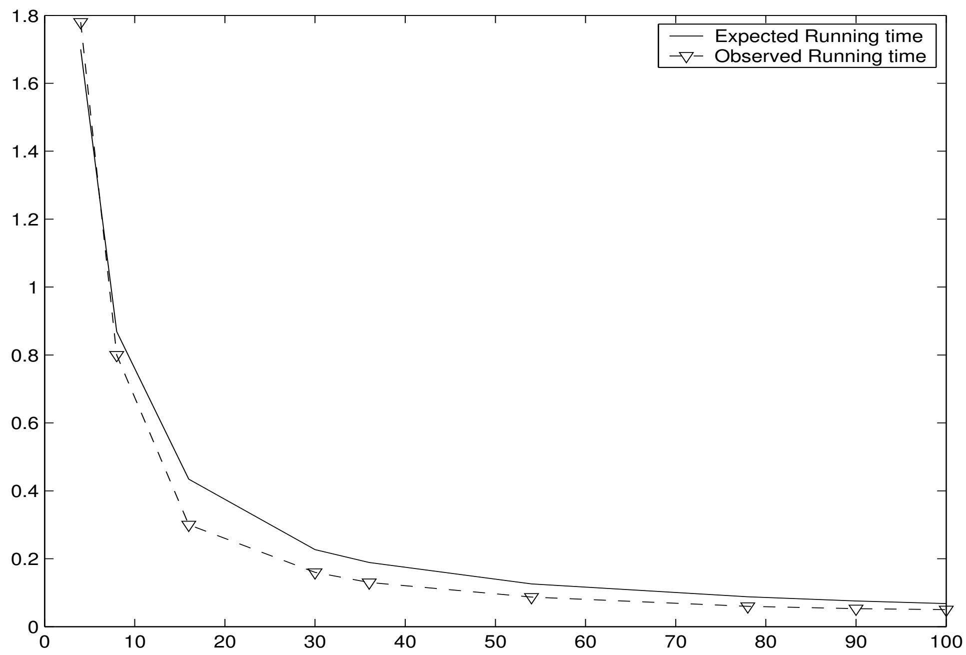
5.5 Design and Coding guidelines

The following guidelines were met while coding and are recommended for further development into the project

1. Percentage of Comments with respect to the code text was ensured to be about 14%
2. Added Function Description for every function
3. The variable declarations had descriptions, limits, if any
4. No one-line if statements in the code
5. The code indented - with or without tabs in f90 - without any tabs in f77 code
6. Dependencies (of change) listed in the documentation
7. Avoided the topology dependent code as far as possible.
8. All functions define the error model (perform input checks) and throw error messages for wrong input.
9. The return status of MPI calls checked at all times.
10. The communication library routines do not call the topology related MPI routines. All such required information passed from the callers.



(a) Predicted vs Observed time spent in distribution of a 4-D array



(b) Predicted vs Observed time spent in computation of the derivative on the 4-D array

Figure 5.4. Predicted vs Observed Execution times for distribution and computation

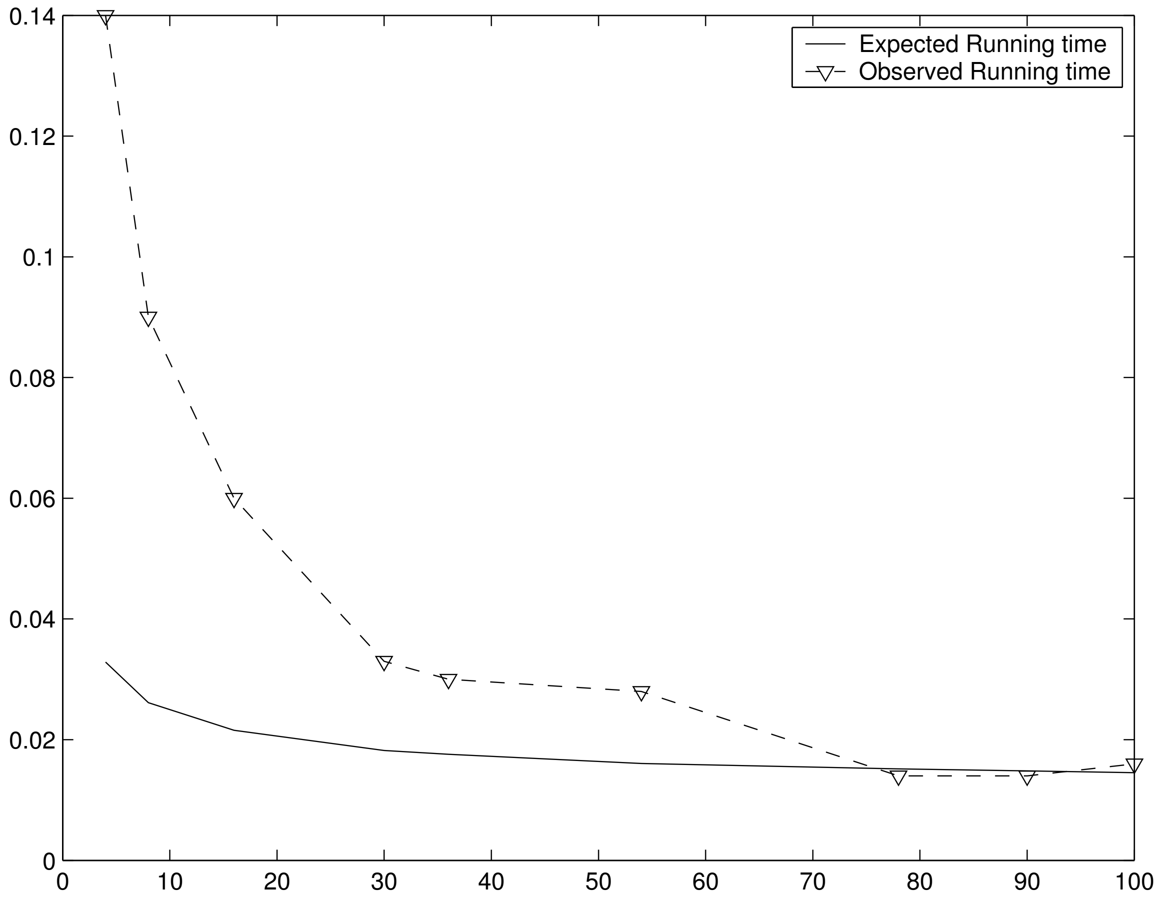


Figure 5.5. Predicted vs Observed Execution times for exchange

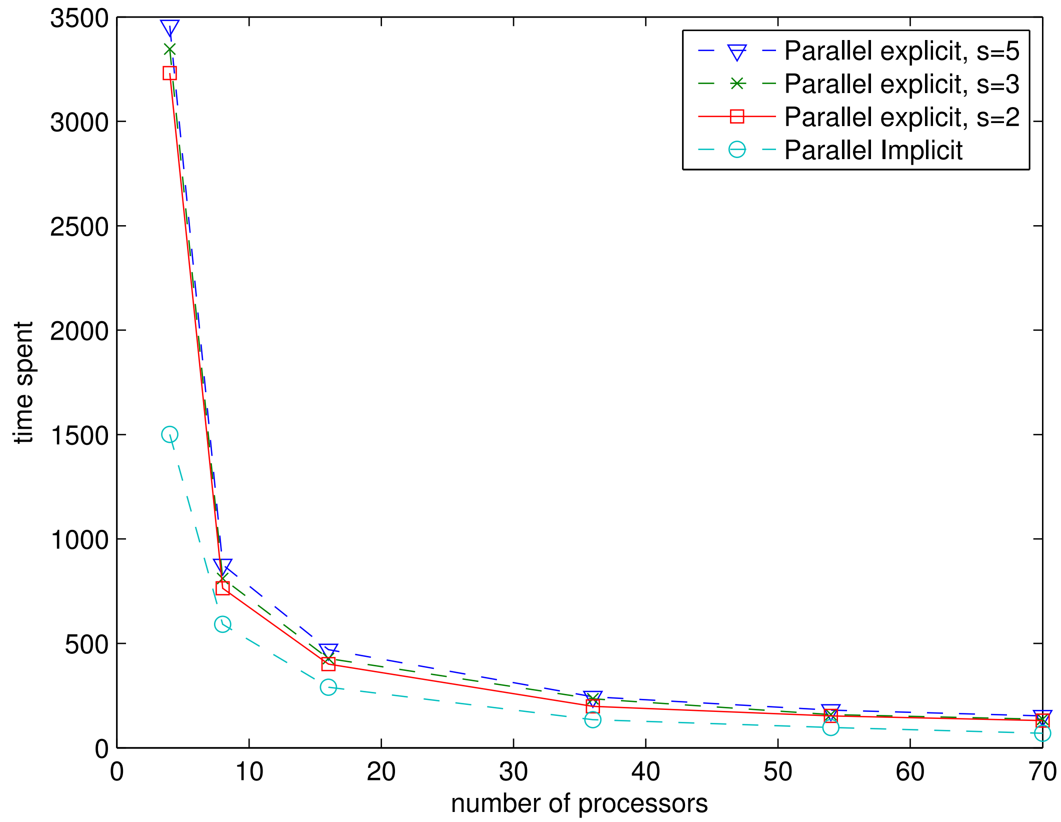


Figure 5.6. Time spent on anantham

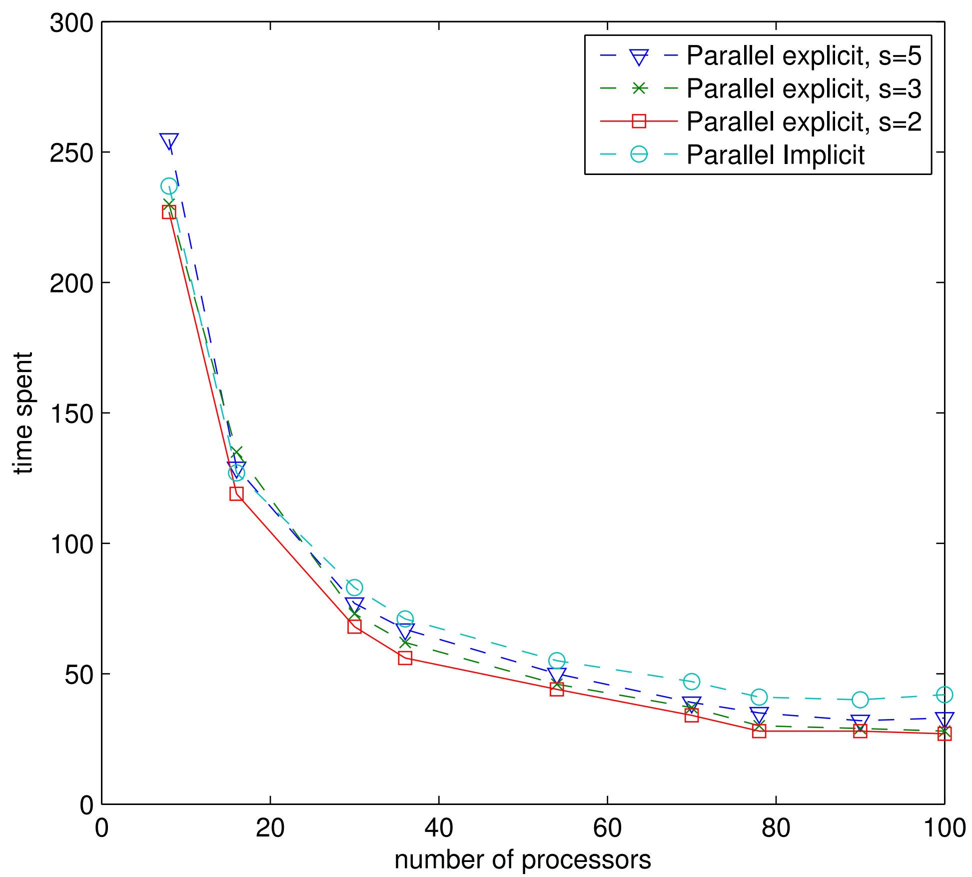


Figure 5.7. Time spent on system-X

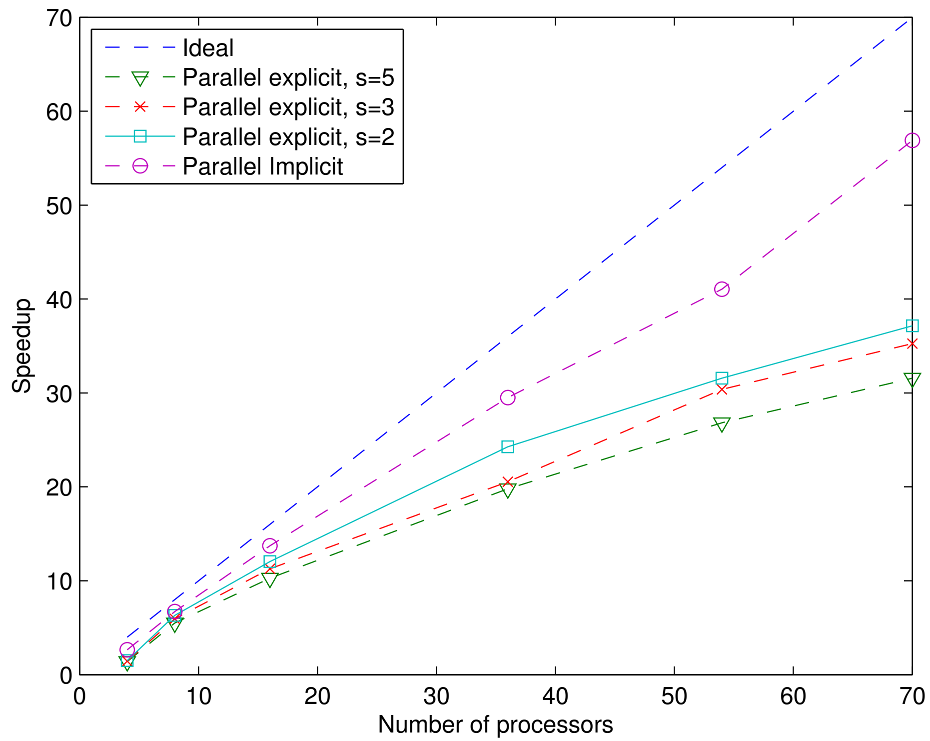


Figure 5.8. Speedup on anantham

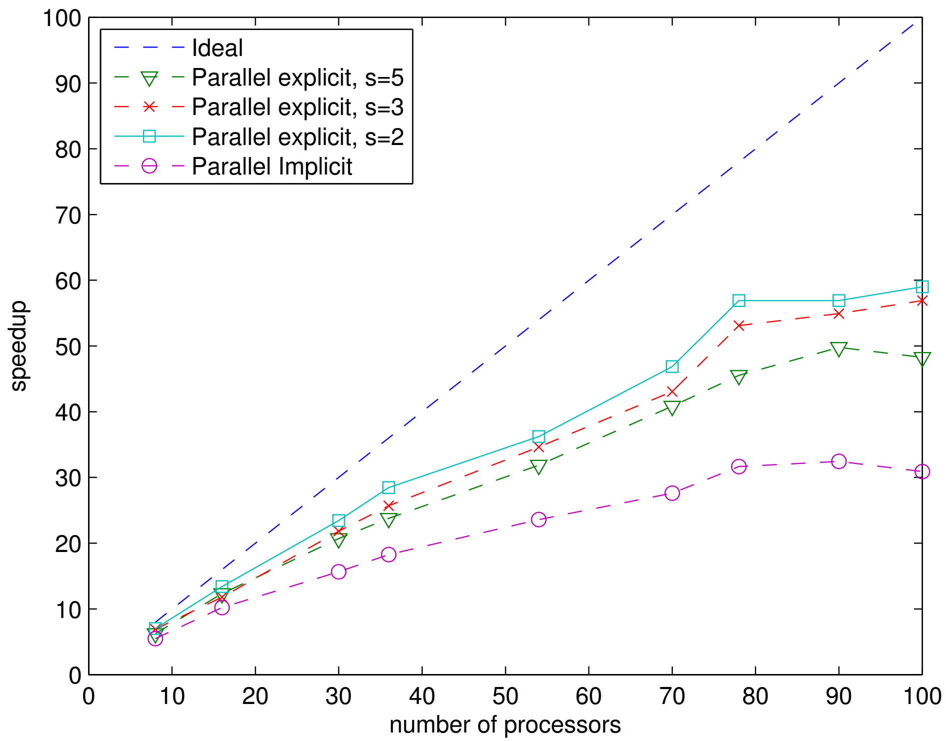


Figure 5.9. Speedup on system-X

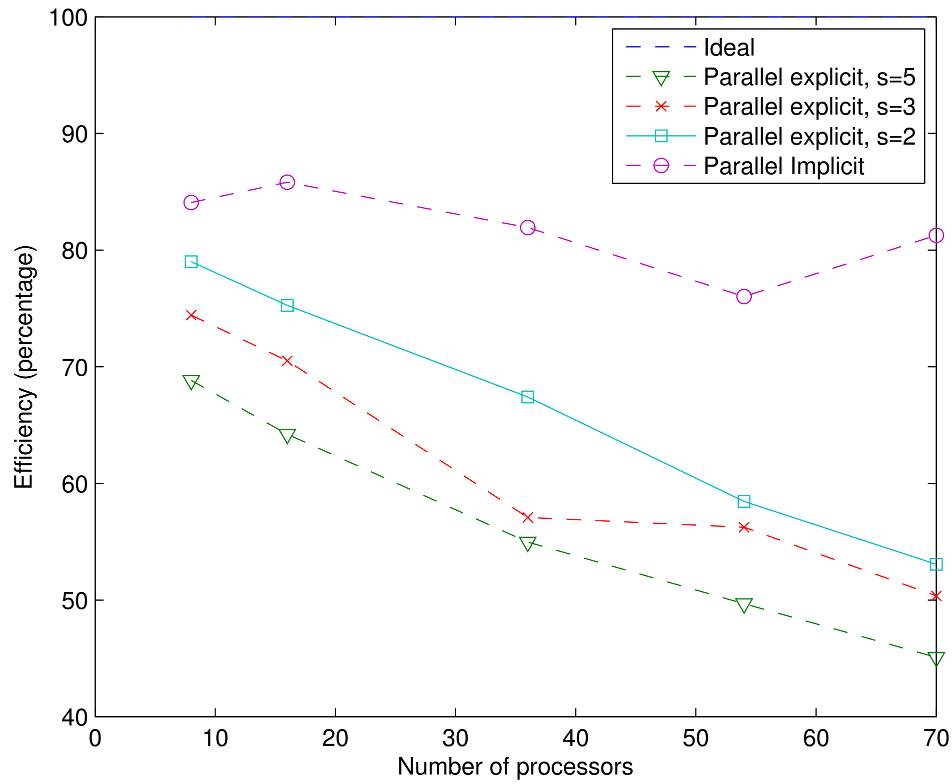


Figure 5.10. Efficiency on anantham

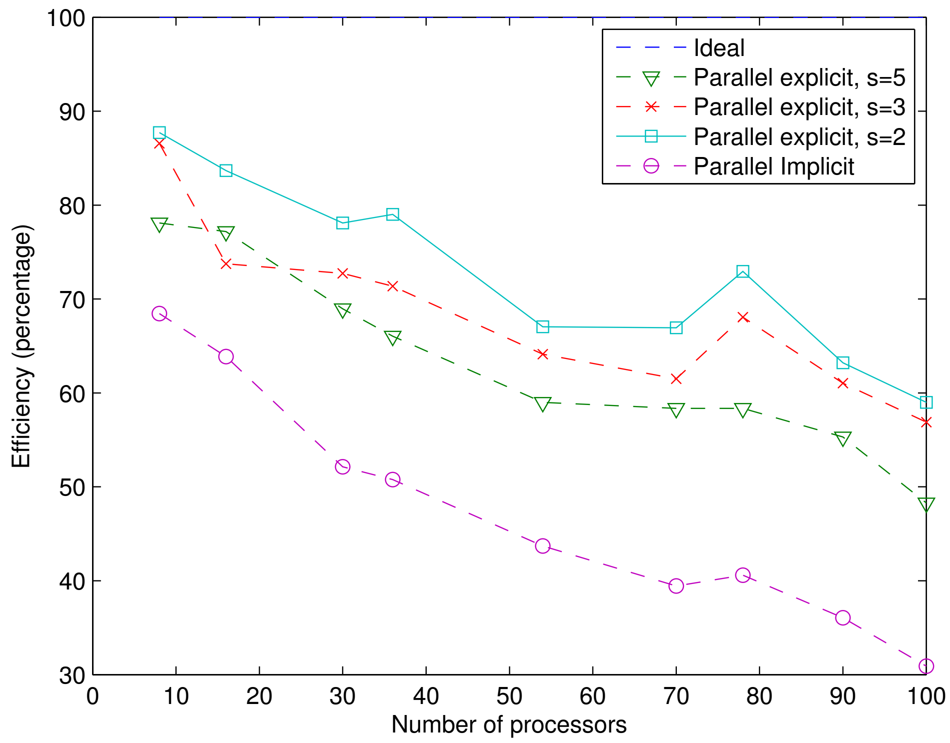
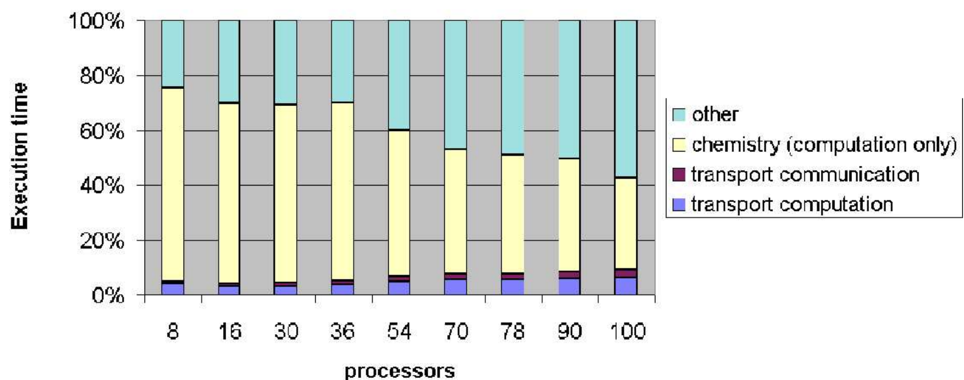
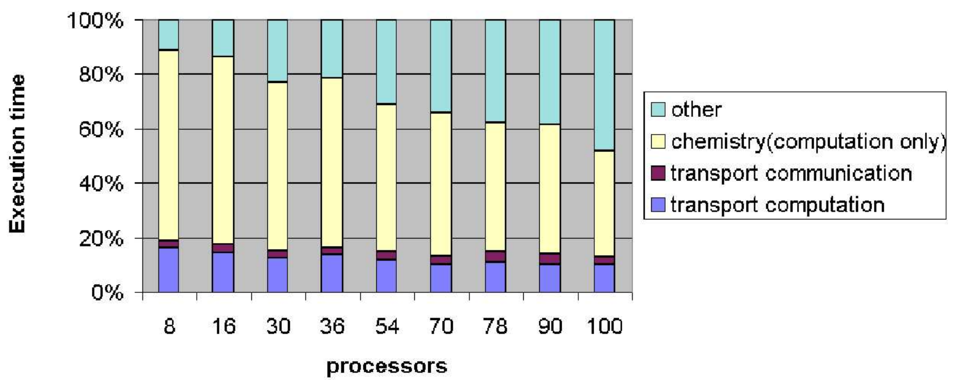


Figure 5.11. Efficiency on system-X



(a) Implicit



(b) Explicit s=5

Figure 5.12. Fractions of Execution Times - Implicit vs Explicit

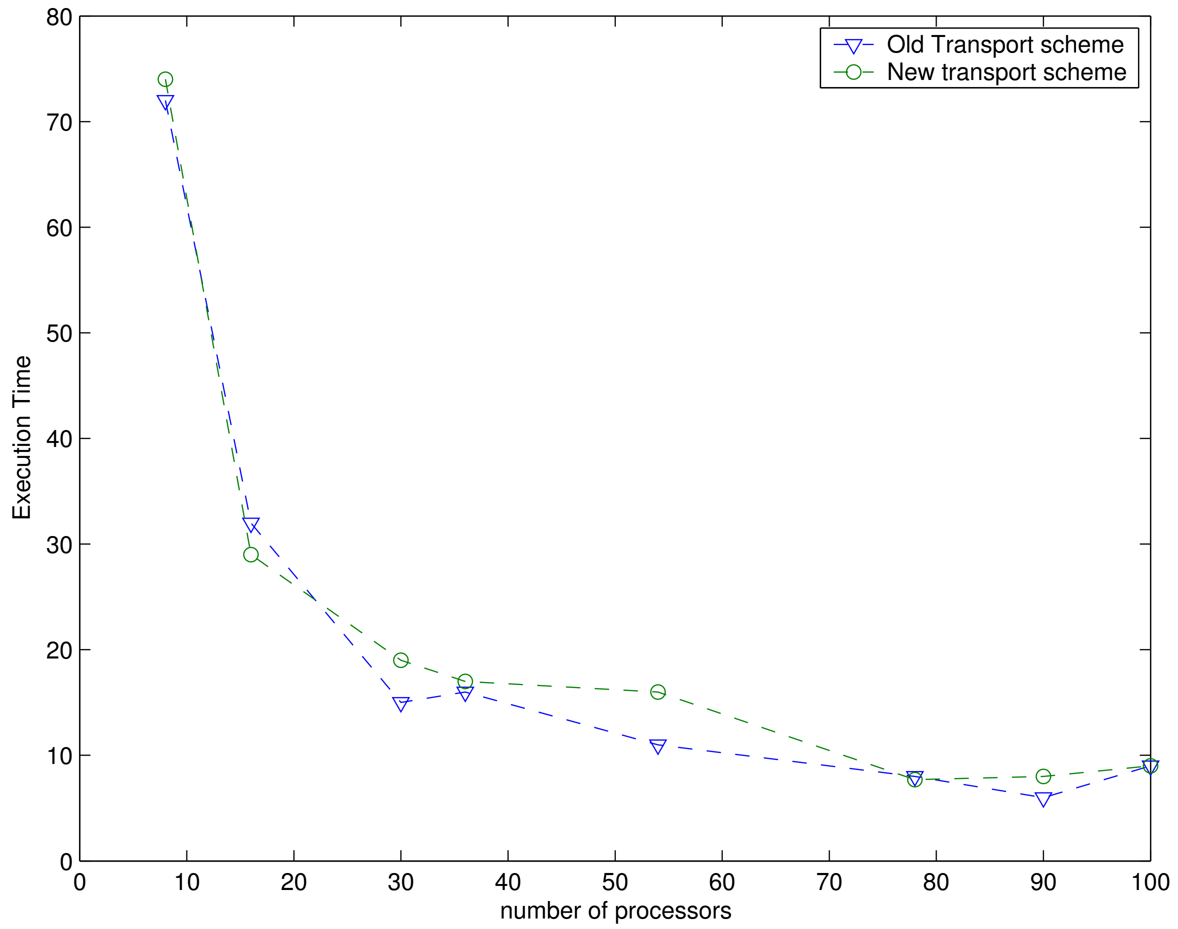


Figure 5.13. Execution times for overlapped computation vs non-overlapped computation scheme

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

This research aimed at assessing the feasibility of explicit methods in the air quality models. With the initial experiments conducted on the moderately stiff problems that the stem model poses, it was found that explicit Runge Kutta Chebyshev methods provide sufficient stability towards the solution of transport equations.

The known advantage with explicit methods is the simpler data dependency which makes one expect the parallelization efficiency of the method using explicit time-stepping to be considerably higher than when using implicit time-stepping.

For the experiments on a real problem, the Runge Kutta Chebyshev methods were implemented in FORTRAN. The H-V partitioning scheme used in the STEM was replaced by a tiled partitioning scheme. The tests were run on two parallel clusters.

It was confirmed that explicit timestepping reporting better efficiency. From the plots in Figure 5.12 it is clear that in case of the explicit time-stepping, the communication time towards transport contributes lesser to the total execution time as the number of processors is increased. With implicit time-stepping on the other hand, the contribution of communication time doesn't decrease noticeably.

The RKC methods exchange data at multiple times per iteration. It was observed that the execution time increases with the number of times the data has to be exchanged among processors (Figure 5.9).

Another important finding was that the load imbalances among processors can badly affect the performance of the methods using explicit time-stepping. The reason is that the methods

using explicit timestepping methods (e.g. the Runge Kutta class of methods experimented with in the work) perform exchange at multiple stages per iteration. This imposes a synchronization of all processors involved in the communication. This makes all the processors wait until the slowest of the processor has returned the computed data.

The above effect can be significant in smaller number of processors, because the difference in time taken for computation between the slowest and fastest processor is higher due to presence of large data per processor. When the number of processors is high, the difference is leveled to a certain extent and parallelization efficiency is closer to what one would expect in a method with explicit time-stepping. However, if the load imbalance among processors is under tolerable limits, the method with explicit time-stepping performs much better than the implicit method, and is more scalable as well.

6.2 Future Work

The foremost activity planned in near future is the replacement of distribution routines by parallel I/O. With parallel I/O, root wouldn't have to distribute the data to every processor. This would result in considerable reduction in the total execution time, especially when the data being used is really large.

The explicit method would be extended to perform adjoint calculations and data assimilations. Specific check-pointing for tiled decomposition is needed for the activity.

The space discretizations techniques would be implemented and tested with larger/shorter stencils. All the experiments in the presented work were run with stencil of size 5. The combined effect of a value of s with a selected stencil width would be studied further.

We plan to have a coupled integration of transport and chemistry which would save the total time spent in computation.

Also, since the current implementation doesn't handle the load imbalance to a satisfactory degree, one of the activities of great importance would be the implementation of adaptive tiling sizes so that the distribution of computation load at every processor is done in a just fashion. This would alleviate the problem of load imbalance resulting in a more efficient parallelization.

APPENDIX A

IMPLEMENTATION DETAILS

A.1 Stem Driver

```
program STEM_DRIVER_PARALLEL

    !

    ! Driver module - Entry to the program

    !

        ! Uses

    use configModule

    use aqcon1

    use aqcon5

    use aqcont

    use aqrxn

    use aqsymb

    use aqspec

    use aqindx

    use species

    use data_io_class

    use memory_allocation_class

    use initial_setup_class

    use debug_class

    use time_management_class

    use unit_conversion_class

    use transport_class

    use transport_class_mf

    use transport_class_fd

    use chemistry_setup_class
```

```
use chemistry_function_class
use domain
use driver
use meteo3d
use depvel
use emarea
use bdf
use tuv
use communication
use computation

    ! Calls
call ASSIGN_AQCON_DATA
call ASSIGN_AQINFO_DATA
call ASSIGN_AQRXN_DATA
call ASSIGN_GAS_SPECIES_INDEXES
call CONVERT_BOUNDARY_UNIT
call CONVERT_EMISSION_UNIT
call Collect4d
call Distribute2d
call Distribute2d_real
call Distribute3d
call Distribute3d_real
call Distribute4D
call Distribute4d
call Distribute4d_real
```

```
call GET_DATE

call GET_STRANG_MDT

call GET_TIME

call INCREASE_TIME_BY_DELTAT

call LOAD_TIME_SPECIFIC_DATA_C

call MPI_BARRIER

call MPI_BCAST

call MPI_COMM_RANK

call MPI_COMM_SIZE

call MPI_Cart_coords

call MPI_Cart_create

call MPI_FINALIZE

call MPI_INIT

call MPI_Recv

call MPI_Send

call MPI_TYPE_COMMIT

call MPI_TYPE_FREE

call MPI_TYPE_HVECTOR

call MPI_TYPE_VECTOR

call MemoryAllocateForBDF

call MemoryAllocateForDEPVEL

call MemoryAllocateForDOMAIN

call MemoryAllocateForDRIVER

call MemoryAllocateForEMAREA

call MemoryAllocateForINITF

call MemoryAllocateForMETEO3D
```

```
call MemoryAllocateForTUV
call OPEN_ALL_INPUT_FILES
call READ_DOMAIN_DATA
call READ_FILE_AQPROPERTY
call READ_GRID_SIZE
call READ_INFO_FILE
call READ_INITF_DATA
call READ_PARALLEL_VARIABLES
call READ_TRANSPORT_VARIABLES
call SET_THE_TIME
call cmm_distrib
call convert_4D
call convert_4Dd
call convert_4d
call doComputation
call dump_r
call exchangeGhostCells4All
call exchangeGhostCells4Kh
call exchangeGridSize
call getBoundaries
call int_distrib1
call int_distrib2
call mpi_type_size
call plot_spc_par
call real_distrib
call rxn
```

```
call sendBoundaries

call set_environment

call stopprog

call tranz_mf_d


    ! Variables

character (len=20) :: fname

integer :: i

integer :: j

integer :: ipsc

integer :: ierr

integer :: ndims

integer, dimension (2) :: dim_size

integer :: wax

integer :: way

integer :: nax

integer :: nay

integer :: nsteps

integer :: sizeofDble

integer :: sizeofReal

integer :: nprocs

integer :: myId

integer :: ax

integer :: ay

integer :: x

integer :: y
```

```
integer, dimension (2) :: coords
integer :: comm_world
integer :: comm_worker
integer :: msg_data
integer :: width

! MPI Datatypes
integer :: Block
integer :: Block2
integer :: Block4
integer :: Block2_n
integer :: Block4_np
integer :: Block4_nl
integer :: r_Block
integer :: r_Block2
integer :: r_Block4
integer :: r_Block2_n
integer :: r_Block4_np
integer :: r_Block4_nl

!
integer, dimension  (MPI_STATUS_SIZE) :: varstatus
logical :: Master
logical, dimension (2) :: periods
logical :: reorder
```

```
real, pointer, dimension (:,:,:,:) :: sl1
real, pointer, dimension (:,:,:,:) :: sp1
real, pointer, dimension (:,:,:,:) :: Concentration
double precision, allocatable, dimension (:,:,:) :: kn
double precision, allocatable, dimension (:,:,:) :: ke
double precision, allocatable, dimension (:,:,:) :: kw
double precision, allocatable, dimension (:,:,:) :: ks
double precision, allocatable, dimension (:,:,:) :: kh_local
double precision, allocatable, dimension (:,:,:) :: kv_local
double precision, allocatable, dimension (:,:,:,:) :: East
double precision, allocatable, dimension (:,:,:,:) :: West
double precision, allocatable, dimension (:,:,:,:) :: South
double precision, allocatable, dimension (:,:,:,:) :: North
double precision, allocatable, dimension (:,:,:,:) :: Conc
double precision, allocatable, dimension (:,:,:,:) :: boundary_X
double precision, allocatable, dimension (:,:,:,:) :: boundary_Y
double precision, allocatable, dimension (:,:,:) :: sz_local
double precision, allocatable, dimension (:,:,:) :: U_local
double precision, allocatable, dimension (:,:,:) :: V_local
double precision, allocatable, dimension (:,:,:) :: W_local
real, allocatable, dimension (:,:) :: Orography_local
real, allocatable, dimension (:,:) :: Longitude_local
real, allocatable, dimension (:,:) :: Latitude_local
real, allocatable, dimension (:,:,:) :: Temperature_local
real, allocatable, dimension (:,:,:) :: cloudWaterContent_local
real, allocatable, dimension (:,:,:) :: rainWaterContent_local
```

```
    real, allocatable, dimension (:,:,:) :: rainVelocity_local
    real, allocatable, dimension (:,:) :: precipitationRate_local
    real, allocatable, dimension (:,:,:) :: cldod_local
    real, allocatable, dimension (:,:) :: kctop_local
    real, allocatable, dimension (:,:,:) :: ccover_local
    real, allocatable, dimension (:,:,:,:) :: sl1_local
    real, allocatable, dimension (:,:,:,:) :: sp1_local
    real, allocatable, dimension (:,:,:,:) :: Conc_real
    real, allocatable, dimension  (:,:,:) :: gridHeight_local_real
    real, allocatable, dimension (:,:) :: dobson_local
    double precision, allocatable, dimension (:,:,:) :: dz_local
    double precision, allocatable, dimension (:,:,:) :: gridHeight_Local
    double precision, allocatable, dimension (:,:,:) :: surfaceEmissionRate_local
    double precision, allocatable, dimension (:,:,:,:) :: elevatedEmission_local
    double precision, allocatable, dimension (:,:,:) :: dryDepositionVelocity_local
    double precision, allocatable, dimension (:,:) :: deltah_local
    double precision :: deltat
    double precision :: deltax
    double precision :: temp_d
    real :: r_deltat

end program STEM_DRIVER_PARALLEL_HV
```

A.2 Module communication

```
module communication

   ! The module contains all the functions needed for communication
   ! between processors (exchnange, broadcast and distribution)

   ! Subroutines and functions

    public subroutine exchangeGhostCells4Kh (id, comm, local_block,&
    ax, ay, nZ, width, N, E, W, S)

    public subroutine exchangeGhostCells4Conc (id, comm, local_block,&
    ax, ay, nZ, nspec, width, N, E, W, S)

    public subroutine exchangeGhostCells4All (id, comm, local_block,&
    ax, ay, nZ, Nspec, width, N, E, W, S)

    public subroutine Distribute4D_Real (comm_worker, Concentration,&
    nX, nY, nZ, nspec, nax, nay, wax, way)

    public subroutine Distribute4D (comm_worker, Concentration,&
    nX, nY, nZ, nspec, nax, nay, wax, way)

    public subroutine collect4D (comm_worker, Concentration, &
    nx, ny, nz, nspec, nax, nay, wax, way)
```

```
public subroutine Distribute3D_Real (comm_worker, A, nX, nY, nZ, &
nax, nay, wax, way)

public subroutine Distribute3D (comm_worker, A, nX, nY, nZ, &
nax, nay, wax, way)

public subroutine Distribute2D_Real (comm_worker, A, nX, nY, &
nax, nay, wax, way)

public subroutine Distribute2D (comm_worker, A, &
nX, nY, nax, nay, wax, way)

public subroutine collect3D (comm_worker, A, nx, ny, nz, &
nax, nay, wax, way)

public subroutine int_distrib1 (id, iend)

public subroutine int_distrib2 (id, numl, nbin, &
ixtrn, iytrn, iztrn, irxng, irxnl, num, mdt, idate, iend)

public subroutine real_distrib (id, ix, iy, iz, is, dx, dy, &
sigmaz, dht, baseh, rmw, dt, ut)

public subroutine getGhostCellVector (x, y, maxX, maxY, &
tN, tE, tW, tS)
```

```
    public subroutine getGhostCellVector_4 (x, y, maxX, maxY, &
    tN, tE, tW, tS)

    public subroutine getBoundaries (id, comm, boundary_X, boundary_Y, &
    ax, ay, nX, nY, nZ, Nspec)

    public subroutine sendBoundaries (comm_worker, &
    boundary_X, nY, boundary_Y, nX, nZ, Nspec, wax, way, nax, nay)

    public subroutine exchangeGridsize (id, comm)

    public subroutine cmm_distrib (id)

    public subroutine FinExchange (id, comm, width, &
    xn, x_start, x_end, yn, y_start, y_end, nz, nspec, &
    Concentration, tn, te, tw, ts, reqn, reqe, reqw, reqs)

    public subroutine InitExchange (id, comm, width, &
    ax, ay, nz, nspec, N, E, W, S, reqn, reqe, reqw, reqs)

end module communication
```

Description of Subroutines and Functions

exchangeGhostCells4Kh

```
    ! The function is responsible for ghost cell exchange for
    ! Kh data

public subroutine exchangeGhostCells4Kh (id, comm, local_block, &
ax, ay, nZ, width, N, E, W, S)

    integer, intent(in) :: id
    integer, intent(in) :: comm
    double precision, intent(in), dimension (aX,aY,nZ) :: local_block
    integer, intent(in) :: ax
    integer, intent(in) :: ay
    integer, intent(in) :: nZ
    integer, intent(in) :: width
    double precision, intent(out), dimension (ay,nZ,width) :: N
    double precision, intent(out), dimension (ax,nZ,width) :: E
    double precision, intent(out), dimension (ax,nZ,width) :: W
    double precision, intent(out), dimension (ay,nZ,width) :: S

    ! Calls: MPI_Cart_coords, MPI_Cart_rank, MPI_Comm_size,
    MPI_ISend, MPI_Recv, MPI_TYPE_COMMIT, MPI_TYPE_FREE,
    MPI_TYPE_HVECTOR, MPI_TYPE_SIZE, MPI_TYPE_VECTOR,
```

```
    MPI_Wait, getGhostCellVector

end subroutine exchangeGhostCells4Kh
```

exchangeGhostCells4Conc

```
    ! The function is responsible for ghost cell exchange for
    ! Concentration matrix

public subroutine exchangeGhostCells4Conc (id, comm, local_block, &
ax, ay, nZ, nspec, width, N, E, W, S)
    integer, intent(in) :: id
    integer, intent(in) :: comm
    double precision, intent(in), dimension (aX,aY,nZ,nspec) :: local_block
    integer, intent(in) :: ax
    integer, intent(in) :: ay
    integer, intent(in) :: nZ
    integer, intent(in) :: nspec
    integer, intent(in) :: width
    double precision, intent(out), dimension (ay,nZ,width,nspec) :: N
    double precision, intent(out), dimension (ax,nZ,width,nspec) :: E
    double precision, intent(out), dimension (ax,nZ,width,nspec) :: W
    double precision, intent(out), dimension (ay,nZ,width,nspec) :: S
    ! Calls: MPI_Cart_coords, MPI_Cart_rank, MPI_Comm_size,
```

```
    MPI_ISend, MPI_Recv, MPI_TYPE_COMMIT, MPI_TYPE_FREE, MPI_TYPE_HVECTOR,
    MPI_TYPE_SIZE, MPI_TYPE_VECTOR, MPI_Wait, getGhostCellVector

end subroutine exchangeGhostCells4Conc
```

exchangeGhostCells4All

```
    ! The function responsible for ghost cell exchange for data
    ! for a situation when root (along with rest of the processors)
    ! needs to receive (updated) data from the neighboring blocks

public subroutine exchangeGhostCells4All (id, comm, local_block, &
ax, ay, nZ, Nspec, width, N, E, W, S)
    integer, intent(in) :: id
    integer, intent(in) :: comm
    double precision, intent(in), dimension (aX,aY,nZ,nspec) :: local_block
    integer, intent(in) :: ax
    integer, intent(in) :: ay
    integer, intent(in) :: nZ
    integer, intent(in) :: Nspec
    integer, intent(in) :: width
    double precision, intent(out), dimension (ay,nZ,width,Nspec) :: N
    double precision, intent(out), dimension (ax,nZ,width,Nspec) :: E
    double precision, intent(out), dimension (ax,nZ,width,Nspec) :: W
```

```
    double precision, intent(out), dimension (ay,nZ,width,Nspec) :: S
    ! Calls: MPI_Cart_coords, MPI_Cart_rank, MPI_Comm_size,
    MPI_ISend, MPI_Recv, MPI_TYPE_COMMIT, MPI_TYPE_FREE, MPI_TYPE_HVECTOR,
    MPI_TYPE_SIZE, MPI_TYPE_VECTOR, MPI_Wait, getGhostCellVector
end subroutine exchangeGhostCells4All
```

Distribute4D_Real

```
    ! Function that partitions 4D-real data among the processors

public subroutine Distribute4D_Real (comm_worker, Concentration, &
nX, nY, nZ, nspec, nax, nay, wax, way)
    integer, intent(in) :: comm_worker
    real, intent(in), dimension (nX,nY,nZ,nspec) :: Concentration
    integer, intent(in) :: nx
    integer, intent(in) :: ny
    integer, intent(in) :: nz
    integer, intent(in) :: nspec
    integer, intent(in) :: nax
    integer, intent(in) :: nay
    integer, intent(in) :: wax
    integer, intent(in) :: way
    ! Calls: MPI_Cart_rank, MPI_Comm_rank, MPI_Send, MPI_TYPE_COMMIT,
    MPI_TYPE_FREE, MPI_TYPE_HVECTOR, MPI_TYPE_VECTOR, mpi_type_size
end subroutine Distribute4D_Real
```

Distribute4D

```
    ! Function that partitions 4D-double precision data among
    ! the processors

public subroutine Distribute4D (comm_worker, Concentration, &
nX, nY, nZ, nspec, nax, nay, wax, way)
    integer, intent(in) :: comm_worker
    double precision, intent(in), dimension (nX,nY,nZ,nspec) :: &
Concentration
    integer, intent(in) :: nx
    integer, intent(in) :: ny
    integer, intent(in) :: nz
    integer, intent(in) :: nspec
    integer, intent(in) :: nax
    integer, intent(in) :: nay
    integer, intent(in) :: wax
    integer, intent(in) :: way

    ! Calls: MPI_Cart_rank, MPI_Comm_rank, MPI_Send, MPI_TYPE_COMMIT,
    MPI_TYPE_FREE, MPI_TYPE_HVECTOR, MPI_TYPE_VECTOR, mpi_type_size
end subroutine Distribute4D
```

collect4D

```
    ! function that collects the processed data sent from the
    ! processors at the root

public subroutine collect4D (comm_worker, Conc, nx, ny, nz, &
nspec, nax, nay, wax, way)
    integer, intent(in) :: comm_worker
    double precision, intent(in), dimension (nx,ny,nz,nspec) :: Conc
    integer, intent(in) :: nX
    integer, intent(in) :: nY
    integer, intent(in) :: nz
    integer, intent(in) :: nspec
    integer, intent(in) :: nax
    integer, intent(in) :: nay
    integer, intent(in) :: wax
    integer, intent(in) :: way

    ! Calls: MPI_Cart_rank, MPI_Comm_rank, MPI_Recv, MPI_TYPE_COMMIT,
    MPI_TYPE_FREE, MPI_TYPE_HVECTOR, MPI_TYPE_VECTOR, mpi_type_size
end subroutine collect4D
```

Distribute3D_Real

```
   ! function that partitions 3-D real data among the processors
```

```
public subroutine Distribute3D_Real (comm_worker, A, nX, nY, nZ,
nax, nay, wax, way)
    integer, intent(in) :: comm_worker
    real, intent(in), dimension (nX,nY,nZ) :: A
    integer, intent(in) :: nx
    integer, intent(in) :: ny
    integer, intent(in) :: nz
    integer, intent(in) :: nax
    integer, intent(in) :: nay
    integer, intent(in) :: wax
    integer, intent(in) :: way

    ! Calls: MPI_Cart_rank, MPI_Comm_rank, MPI_Send, MPI_TYPE_COMMIT,
    MPI_TYPE_FREE, MPI_TYPE_HVECTOR, MPI_TYPE_VECTOR, mpi_type_size
end subroutine Distribute3D_Real
```

Distribute3D

```
    ! function that partitions 3D double precision data
    ! among the processors
public subroutine Distribute3D (comm_worker, A, nX, nY, nZ,
nax, nay, wax, way)
    integer, intent(in) :: comm_worker
    double precision, intent(in), dimension (nX,nY,nZ) :: A
```

```
    integer, intent(in) :: nx
    integer, intent(in) :: ny
    integer, intent(in) :: nz
    integer, intent(in) :: nax
    integer, intent(in) :: nay
    integer, intent(in) :: wax
    integer, intent(in) :: way
    ! Calls: MPI_Cart_rank, MPI_Comm_rank, MPI_Send, MPI_TYPE_COMMIT,
    MPI_TYPE_FREE, MPI_TYPE_HVECTOR, MPI_TYPE_VECTOR, mpi_type_size
end subroutine Distribute3D
```

Distribute2D_Real

```
   ! function that partitions 2D real data among the processors
public subroutine Distribute2D_Real (comm_worker, A, nX, nY, nax,
nay, wax, way)
    integer, intent(in) :: comm_worker
    real, intent(in), dimension (nX,nY) :: A
    integer, intent(in) :: nx
    integer, intent(in) :: ny
    integer, intent(in) :: nax
    integer, intent(in) :: nay
    integer, intent(in) :: wax
    integer, intent(in) :: way
    ! Calls: MPI_Cart_rank, MPI_Comm_rank, MPI_Send, MPI_TYPE_COMMIT,
```

```
    MPI_TYPE_FREE, MPI_TYPE_VECTOR, mpi_type_size
end subroutine Distribute2D_Real
```

Distribute2D

```
    ! function that partitions 4-D double precision data among
    ! processors
public subroutine Distribute2D (comm_worker, A, nX, nY,
nax, nay, wax, way)
    integer, intent(in) :: comm_worker
    double precision, intent(in), dimension (nX,nY) :: A
    integer, intent(in) :: nx
    integer, intent(in) :: ny
    integer, intent(in) :: nax
    integer, intent(in) :: nay
    integer, intent(in) :: wax
    integer, intent(in) :: way
    ! Calls: MPI_Cart_rank, MPI_Comm_rank, MPI_Send, MPI_TYPE_COMMIT,
    MPI_TYPE_FREE, MPI_TYPE_VECTOR, mpi_type_size
end subroutine Distribute2D
```

collect3D

```
    ! function that collects processed 3-D data at the root sent from
```

```
    ! the rest of the processors


public subroutine collect3D (comm_worker, A, nx, ny, nz, &
nax, nay, wax, way)
    integer, intent(in) :: comm_worker
    double precision, intent(in), dimension (nx,ny,nz) :: A
    integer, intent(in) :: nX
    integer, intent(in) :: nY
    integer, intent(in) :: nz
    integer, intent(in) :: nax
    integer, intent(in) :: nay
    integer, intent(in) :: wax
    integer, intent(in) :: way
    ! Calls: MPI_Cart_rank, MPI_Comm_rank, MPI_Recv, MPI_TYPE_COMMIT,
    MPI_TYPE_FREE, MPI_TYPE_HVECTOR, MPI_TYPE_VECTOR
end subroutine collect3D
```

int_distrib1

```
    ! broadcasts integer data among the processors
public subroutine int_distrib1 (id, iend)
    integer, intent(in) :: id
    integer, intent(inout) :: iend
    ! Calls: MPI_BCAST
end subroutine int_distrib1
```

int_distrib2

```
    ! broadcasts integer data among the processors
public subroutine int_distrib2 (id, numl, nbin,
ixtrn, iytrn, iztrn, irxng, irxnl, num, mdt, idate, iend)
    integer, intent(in) :: id
    integer, dimension (3,4) :: numl
    integer :: nbin
    integer :: ixtrn
    integer :: iytrn
    integer :: iztrn
    integer :: irxng
    integer :: irxnl
    integer :: num
    integer :: mdt
    integer, dimension (3) :: idate
    integer :: iend
    ! Calls: MPI_BCAST
end subroutine int_distrib2
```

real_distrib

```
    ! broadcasts real data among the processors
public subroutine real_distrib (id, ix, iy, iz, is, dx, dy,
```

```
sigmaz, dht, baseh, rmw, dt, ut)
    integer, intent(in) :: id
    integer :: ix
    integer :: iy
    integer :: iz
    integer :: is
    real, dimension (ix) :: dx
    real, dimension (iy) :: dy
    real, dimension (iz) :: sigmaz
    real :: dht
    real :: baseh
    real, dimension (is) :: rmw
    real :: dt
    real :: ut
    ! Calls: MPI_BCAST
end subroutine real_distrib
```

getGhostCellVector

```
    ! function obtains the (N,E,W,S) vector needed for block exchange
public subroutine getGhostCellVector (x, y, maxX, maxY, tN, tE, tW, tS)
    integer, intent(in) :: x
    integer, intent(in) :: y
    integer, intent(in) :: maxX
    integer, intent(in) :: maxY
```

```
    logical, intent(out) :: tN
    logical, intent(out) :: tE
    logical, intent(out) :: tW
    logical, intent(out) :: tS
end subroutine getGhostCellVector
```

getBoundaries

```
    ! function obtains boundary data from the root at all the
    ! processors that own the sub-domain along the boundaries
public subroutine getBoundaries (id, comm, boundary_X, boundary_Y,
ax, ay, nX, nY, nZ, Nspec)
    integer, intent(in) :: id
    integer, intent(in) :: comm
    double precision, intent(out), dimension (ay,nZ,2,Nspec) :: boundary_X
    double precision, intent(out), dimension (ax,nZ,2,Nspec) :: boundary_Y
    integer, intent(in) :: ax
    integer, intent(in) :: ay
    integer, intent(in) :: nX
    integer, intent(in) :: nY
    integer, intent(in) :: nZ
    integer, intent(in) :: Nspec
    ! Calls: MPI_Cart_coords, MPI_Comm_rank, MPI_Comm_size,
    MPI_Recv, MPI_TYPE_COMMIT, MPI_TYPE_FREE,
    MPI_TYPE_HVECTOR, MPI_TYPE_SIZE, MPI_TYPE_VECTOR
```

```
end subroutine getBoundaries
```

sendBoundaries

```
    ! function sends data from root to all processors that own
    ! a subdomain along the boundary
public subroutine sendBoundaries (comm_worker,
boundary_X, nY, boundary_Y, nX, nZ, Nspec, wax, way, nax, nay)
    integer, intent(in) :: comm_worker
    double precision, intent(in), dimension (nY,nZ,2,Nspec) :: boundary_X
    integer, intent(in) :: nY
    double precision, intent(in), dimension (nX,nZ,2,Nspec) :: boundary_Y
    integer, intent(in) :: nX
    integer, intent(in) :: nZ
    integer, intent(in) :: Nspec
    integer, intent(in) :: wax
    integer, intent(in) :: way
    integer, intent(in) :: nax
    integer, intent(in) :: nay
    ! Calls: MPI_Cart_rank, MPI_Send, MPI_TYPE_COMMIT,
    MPI_TYPE_FREE, MPI_TYPE_HVECTOR, MPI_TYPE_SIZE,
    MPI_TYPE_VECTOR, mpi_comm_rank, mpi_comm_size
end subroutine sendBoundaries
```

cmm_distrib

```
    ! broadcasts cmm data among the processors
public subroutine cmm_distrib (id)
    integer, intent(in) :: id
    ! Calls: MPI_BCAST
end subroutine cmm_distrib
```

InitExchange

```
    ! Function initializes the exchange by performing isends
    ! and returns

public subroutine InitExchange (id, comm, width, &
ax, ay, nz, nspec, N, E, W, S, reqn, reqe, reqw, reqs)
    integer, intent(in) :: id
    integer, intent(in) :: comm
    integer, intent(in) :: width
    integer, intent(in) :: ax
    integer, intent(in) :: ay
    integer, intent(in) :: nz
    integer, intent(in) :: nspec
    double precision, intent(in), dimension (ay,nZ,width,Nspec) :: N
    double precision, intent(in), dimension (ax,nZ,width,Nspec) :: E
    double precision, intent(in), dimension (ax,nZ,width,Nspec) :: W
```

```
    double precision, intent(in), dimension (ay,nZ,width,Nspec) :: S
    integer, intent(out) :: reqn
    integer, intent(out) :: reqe
    integer, intent(out) :: reqw
    integer, intent(out) :: reqs
    ! Calls: MPI_Cart_coords, MPI_Cart_rank, MPI_Comm_size, &
MPI_ISend, MPI_TYPE_COMMIT, MPI_TYPE_EXTENT, &
MPI_TYPE_HVECTOR, MPI_TYPE_VECTOR, getGhostCellVector
end subroutine InitExchange
```

FinExchange

```
    ! The functions performs waits on the Isends made for the
    ! block exchange

public subroutine FinExchange (id, comm, width, &
xn, x_start, x_end, yn, y_start, y_end, nz, nspec, &
Concentration, tn, te, tw, ts, reqn, reqe, reqw, reqs)

    integer, intent(in) :: id
    integer, intent(in) :: comm
    integer, intent(in) :: width
    integer, intent(in) :: xn
    integer, intent(in) :: x_start
    integer, intent(in) :: x_end
```

```
    integer, intent(in) :: yn
    integer, intent(in) :: y_start
    integer, intent(in) :: y_end
    integer, intent(in) :: nz
    integer, intent(in) :: nspec
    double precision, intent(inout), &
dimension (xn,yn,nz,nspec) :: Concentration
    logical, intent(in) :: tn
    logical, intent(in) :: te
    logical, intent(in) :: tw
    logical, intent(in) :: ts
    integer, intent(in) :: reqn
    integer, intent(in) :: reqe
    integer, intent(in) :: reqw
    integer, intent(in) :: reqs
    ! Calls: MPI_Cart_coords, MPI_Cart_rank, MPI_Recv, MPI_TYPE_COMMIT,&
 MPI_TYPE_EXTENT, MPI_TYPE_HVECTOR, MPI_TYPE_VECTOR, MPI_Wait
end subroutine FinExchange
```

A.3 Module Computation

```
module computation
    ! Module contains all the funcitons required to perform
    ! advection-diffusion computations

        ! Uses
    use communication

        ! Subroutines and functions
    public subroutine doComputation (id, comm, A, UU, VV, KKh, &
    sx, sy, North, kN, East, ke, West, kw, South, ks, w, &
    ix, iy, nZ, Nspec, iair, deltat, deltax, nsteps)

    public subroutine SolveRKC2dBlock (id, comm, nZ, Nspec, &
    width, x_start, x_end, xN, y_start, y_end, yN, UU, VV, &
    kkh, air, sx, sy, time_interval, dx, dt, Cin, Cout, tn, te, tw, ts)

    public subroutine CalcBlockDerivative1pN (id, comm, nZ, Nspec, &
    width, x_start, x_end, xn, y_start, y_end, yN, &
    Concentration, UU, VV, kh, air, dx, &
    boundary_X, boundary_Y, Derivative4D, tN, tE, tW, tS)

    public subroutine CalcPeriD1p (id, nz, nspec, w, &
    x_start, x_end, xn, y_start, y_end, yN, &
    Concentration, UU, VV, kh, air, dx, &
```

```
    boundary_X, boundary_Y, Derivative4D, tn, te, tw, ts)

    public subroutine CalcCoreD1p (id, nz, nspec, width, &
    x_start, x_end, xn, y_start, y_end, yN, &
    Concentration, Derivative4d, UU, VV, kh, air, dx)

    public subroutine CalcBlockDerivative1p (id, comm, nZ, Nspec, &
    iair, width, x_start, x_end, xn, y_start, y_end, yN, &
    Concentration, UU, VV, kh, dx, boundary_X, boundary_Y, &
    Derivative4D, tN, tE, tW, tS)

    public subroutine getMarkers (tN, tE, tW, tS, ix, iy, w, &
    x_start, x_end, xN, y_start, y_end, yN)

end module computation
```

Description of Subroutines and Functions

doComputation

```
   ! The function updates the concentration matrix by integrating
   ! using an explicit technique

public subroutine doComputation (id, comm, A, UU, VV, KKh, &
sx, sy, North, kN, East, ke, West, kw, South, ks, w, &
ix, iy, nZ, Nspec, iair, deltat, deltax, nsteps)
```

```
integer, intent(in) :: id
integer, intent(in) :: comm
double precision, intent(inout), dimension (ix,iy,nZ,Nspec) :: A
double precision, intent(in), dimension (ix,iy,nZ) :: UU
double precision, intent(in), dimension (ix,iy,nZ) :: VV
double precision, intent(in), dimension (ix,iy,nZ) :: Kkh
double precision, intent(in), dimension (iy,nZ,2,Nspec) :: sx
double precision, intent(in), dimension (ix,nZ,2,Nspec) :: sy
double precision, intent(in), dimension (iy,nZ,w,Nspec) :: North
double precision, intent(in), dimension (iy,nZ,w) :: kn
double precision, intent(in), dimension (ix,nZ,w,Nspec) :: East
double precision, intent(in), dimension (ix,nz,w) :: ke
double precision, intent(in), dimension (ix,nZ,w,Nspec) :: West
double precision, intent(in), dimension (ix,nZ,w) :: kw
double precision, intent(in), dimension (iy,nZ,w,Nspec) :: South
double precision, intent(in), dimension (iy,nZ,w) :: ks
integer, intent(in) :: w
integer, intent(in) :: ix
integer, intent(in) :: iy
integer, intent(in) :: nZ
integer, intent(in) :: Nspec
integer, intent(in) :: iair
double precision, intent(in) :: Deltat
double precision, intent(in) :: Deltax
integer :: Nsteps
```

```
    ! Calls: MPI_Cart_coords, SolveRKC2dBlock, &
    getGhostCellVector, getMarkers, mpi_comm_size
end subroutine doComputation
```

SolveRKC2dBlock

```
    ! The function implements the Runge Kutta Chebyshev method
    ! for integration of differential equations

public subroutine SolveRKC2dBlock (id, comm, nZ, Nspec, width, &
x_start, x_end, xN, y_start, y_end, yN, &
UU, VV, kkh, air, sx, sy, time_i        nterval, &
dx, dt, Cin, Cout, tn, te, tw, ts)

    integer, intent(in) :: id
    integer, intent(in) :: comm
    integer, intent(in) :: nZ
    integer, intent(in) :: Nspec
    integer, intent(in) :: width
    integer, intent(in) :: x_start
    integer, intent(in) :: x_end
    integer, intent(in) :: xN
    integer, intent(in) :: y_start
    integer, intent(in) :: y_end
    integer, intent(in) :: yN
```

```
    double precision, intent(in), &
dimension (x_end-x_start+1,y_end-y_start+1,nZ) :: UU
    double precision, intent(in), &
dimension (x_end-x_start+1,y_end-y_start+1,nZ) :: VV
    double precision, intent(in), dimension (xn,yn,nz) :: kkh
    double precision, intent(in), dimension (xn,yn,nz) :: air
    double precision, intent(in), &
 dimension (y_end-y_start+1,nZ,2,Nspec) :: sx
    double precision, intent(in), &
 dimension (x_end-x_start+1,nZ,2,Nspec) :: sy
    double precision, intent(in) :: time_interval
    double precision, intent(in) :: dx
    double precision, intent(in) :: dt
    double precision, intent(in),  dimension (xn,yn,nZ,Nspec) :: Cin
    double precision, intent(out), dimension (xn,yn,nZ,Nspec) :: Cout
    logical, intent (in) :: tn
    logical, intent (in) :: te
    logical, intent (in) :: tw
    logical, intent (in) :: ts
    ! Calls: CalcBlockDerivative1pN, calcT, calcT_p, calcT_pp
end subroutine SolveRKC2dBlock
```

CalcBlockDerivative1pN

```
   ! Calcualtes the derivative for the whole subdomain at a block
```

```
   ! and tries to overlap communication time and computation time

public subroutine CalcBlockDerivative1pN (id, comm, nZ, Nspec, &
width, x_start, x_end, xn, y_start, y_end, yN, &
Concentration, UU, VV, kh, air, dx, boundary_X, boundary_Y, &
Derivative4D, tN, tE, tW, tS)
    integer, intent(in) :: id
    integer, intent(in) :: comm
    integer, intent(in) :: nz
    integer, intent(in) :: nspec
    integer, intent(in) :: width
    integer, intent(in) :: x_start
    integer, intent(in) :: x_end
    integer, intent(in) :: xn
    integer, intent(in) :: y_start
    integer, intent(in) :: y_end
    integer, intent(in) :: yn
    double precision, intent(in), &
dimension (xn,yn,nz,nspec) :: Concentration
    double precision, intent(in), &
dimension (x_end-x_start+1,y_end-y_start+1,nz) :: UU
    double precision, intent(in), &
dimension (x_end-x_start+1,y_end-y_start+1,nz) :: VV
    double precision, intent(in), dimension (xn,yn,nz) :: Kh
    double precision, intent(in), dimension (xn,yn,nz) :: air
    double precision, intent(in) :: dx
```

```
    double precision, intent(in), &
dimension (y_end-y_start+1,nz,2,nspec) :: boundary_X
    double precision, intent(in), &
dimension (x_end-x_start+1,nz,2,nspec) :: boundary_Y
    double precision, intent(out), &
dimension (xn,yn,nz,nspec) :: Derivative4d
    logical, intent(in) :: tn
    logical, intent(in) :: te
    logical, intent(in) :: tw
    logical, intent(in) :: ts
    ! Calls: CalcCoreD1p, CalcPeriD1p, FinExchange, InitExchange

end subroutine CalcBlockDerivative1pN
```

CalcPeriD1p

```
    ! Calculates only the part of the sub-domain which is to be
    ! exchanged with neighboring processors
    ! This is used for overlapping communication and computation

public subroutine CalcPeriD1p (id, nz, nspec, w, &
  x_start, x_end, xn, y_start, y_end, yN,&
Concentration, UU, VV, kh, air, dx, &
boundary_X, boundary_Y, Derivative4D, tn, te, tw, ts)
    integer, intent(in) :: id
```

```
    integer, intent(in) :: nz
    integer, intent(in) :: nspec
    integer, intent(in) :: w
    integer, intent(in) :: x_start
    integer, intent(in) :: x_end
    integer, intent(in) :: xn
    integer, intent(in) :: y_start
    integer, intent(in) :: y_end
    integer, intent(in) :: yn
    double precision, intent(in), &
dimension  (xn,yn,nz,nspec) :: Concentration
    double precision, intent(in),&
  dimension (x_end-x_start+1,y_end-y_start+1,nz) :: UU
    double precision, intent(in), &
dimension (x_end-x_start+1,y_end-y_start+1,nz) :: VV
    double precision, intent(in), dimension (xn,yn,nz) :: Kh
    double precision, intent(in), dimension  (xn,yn,nz):: ai
    double precision, intent(in), &
dimension (y_end-y_start+1,nz,2,nspec) :: boundary_X
    double precision, intent(in), &
dimension (x_end-x_start+1,nz,2,nspec) :: boundary_Y
    double precision, intent(out), &
dimension (xn,yn,nz,nspec) :: Derivative4D
    logical, intent(in) :: tn
    logical, intent(in) :: te
    logical, intent(in) :: tw
```

```
    logical, intent(in) :: ts
end subroutine CalcPeriD1p
```

CalcCoreD1p

```
    ! function calculates the part of the sub-domain which is
    ! not to be exchanged with the neighboring processors
    ! This is used for overlapping communication and computation

public subroutine CalcCoreD1p (id, nz, nspec, width, &
x_start, x_end, xn, y_start, y_end, yN, &
Concentration, Derivative4d, UU, VV, kh, air, dx)
    integer, intent(in) :: id
    integer, intent(in) :: nz
    integer, intent(in) :: nspec
    integer, intent(in) :: width
    integer, intent(in) :: x_start
    integer, intent(in) :: x_end
    integer, intent(in) :: xn
    integer, intent(in) :: y_start
    integer, intent(in) :: y_end
    integer, intent(in) :: yn
    double precision, intent(in), &
dimension (xn,yn,nz,nspec) :: Concentration
    double precision, intent(inout), &
```

```
dimension  (xn,yn,nz,nspec) :: Derivative4d
    double precision, intent(in), &
dimension (x_end-x_start+1,y_end-y_start+1,nz) :: UU
    double precision, intent(in), &
dimension (x_end-x_start+1,y_end-y_start+1,nz) :: VV
    double precision, intent(in), dimension (xn,yn,nz) :: Kh
    double precision, dimension (xn,yn,nz) :: air
    double precision, intent(in) :: dx
end subroutine CalcCoreD1p
```

CalcBlockDerivative1p

```
   ! Calculates the derivative for the whole subdomain (4-D)

public subroutine CalcBlockDerivative1p (id, comm, nZ, Nspec, &
width, x_start, x_end, xn, y_start, y_end, yN, &
Concentration, UU, VV, kh, dx, boundary_X, boundary_Y, &
Derivative4D, tN, tE, tW, tS)

    integer, intent(in) :: id
    integer, intent(in) :: comm
    integer, intent(in) :: nZ
    integer, intent(in) :: Nspec
    integer, intent(in) :: width
    integer, intent(in) :: x_start
```

```
    integer, intent(in) :: x_end
    integer, intent(in) :: xN
    integer, intent(in) :: y_start
    integer, intent(in) :: y_end
    integer, intent(in) :: yN
    double precision, intent(in), &
dimension (xn,yn,nZ,Nspec) :: Concentration
    double precision, intent(in), &
dimension (x_end-x_start+1,y_end-y_start+1,nZ) :: UU
    double precision, intent(in), &
dimension  (x_end-x_start+1,y_end-y_start+1,nZ) :: VV
    double precision, intent(in), dimension  (xn,yn,nZ) :: Kh
    double precision, intent(in), dimension  (xn,yn,nZ) :: Air
    double precision, intent(in) :: dx
    double precision, intent(in), &
dimension (y_end-y_start+1,nZ,2,Nspec) :: boundary_X
    double precision, intent(in), &
dimension (x_end-x_start+1,nZ,2,Nspec) :: boundary_Y
    double precision, intent(out), &
dimension (xn,yn,nZ,Nspec) :: Derivative4D
    logical, intent(in) :: tN
    logical, intent(in) :: tE
    logical, intent(in) :: tW
    logical, intent(in) :: tS
    ! Calls: exchangeGhostcells4All
end subroutine CalcBlockDerivative1p
```

getMarkers

```
    ! function obtains markers for the enhanced matrixed
    ! used for computation of the derivative

public subroutine getMarkers (tN, tE, tW, tS, ix, iy, w, &
x_start, x_end, xN, y_start, y_end, yN)
    logical, intent(in) :: tN
    logical, intent(in) :: tE
    logical, intent(in) :: tW
    logical, intent(in) :: tS
    integer, intent(in) :: ix
    integer, intent(in) :: iy
    integer, intent(in) :: w
    integer, intent(out) :: x_start
    integer, intent(out) :: x_end
    integer, intent(out) :: xN
    integer, intent(out) :: y_start
    integer, intent(out) :: y_end
    integer, intent(out) :: yN
end subroutine getMarkers
```

BIBLIOGRAPHY

[1] Assyr Abdulle, and Alexei A. Medovikov, "Second order Chebyshev methods based on orthogonal polynomials,"*Numer. Math.*, Issue 90, Vol.1,pp. 1-18, 2001

[2] J.G. Verwer, B.P. Sommeijer, W.Hundsdorfer, "RKC Time-Stepping for Advection-Diffusion-Reaction Problems," *J. Comput. Physics*, Issue 201, pp. 61-79, 2004

[3] J.G. Verwer, "Runge Kutta Method for parabolic partial differential equations," *Applied Numerical Mathematics*, Issue 22, pp. 359-379, 1996

[4] Z. Zheng and L. Petzold, "Runge-Kutta Chebyshev Projection Method," *J. Comput. Physics*, 2005

[5] Jaideep Ray, C.A. Kennedy, Sophia Lefantzi and H.N. Najm, "High-order spatial discretizations and extended stability methods for reacting flows on structured adaptively refined meshes," *Proceedings of Third Joint Meeting of the U.S. Sections of the Combustion Institute Chicago, USA*, March 2003

[6] Philipp Miehe, Adrian Sandu, Gregory R. Carmichael, Youhua Tang and Dacian Daescu, "A communication library for the parallelization of air quality models on structured grids," *Atmospheric Environment*, Volume 36, Issue 24, pp. 3917-3930, 2002

[7] A. Sandu, D.N. Daescu, G.R. Carmichael, and T. Chai, "Adjoint Sensitivity Analysis of Regional Air Quality Models", *J. Comput. Physics*, Vol. 204, p. 222-252, 2005

[8] W. Owczarz and Z. Zlatev,"Running a large air pollution model on an IBM SMP computer," *International Journal on Computer Research*, Vol. 10, No. 4, pp. 321-330, 2001

[9] H. Elbern, "Parallelization and load balancing of a comprehensive atmospheric chemistry transport model," *Atm. Env.* 31, pp. 3561-3574, 1997

[10] H. Elbern, "On the load balancing problem of comprehensive air quality models," *SAMS*, 32, pp. 3156, 1998

[11] D. Dabdub and R. Manohar, "Peformance and portability of air quality model," *Parallel Computing*, Issue 23, Vol. 14, pp. 2187-2200, 1997

[12] Encyclopædia Britannica, " Atmosphere ," *Encyclopædia Britannica Online*, 2424, June 2006

[13] http://www.tcf.vt.edu

[14] http://www.openpbs.org

[15] http://www.mpi-forum.org

[16] http://www.unidata.ucar.edu/software/netcdf/

[17] http://hdf.ncsa.uiuc.edu/HDF5/

[18] http://www.netlib.org/blas/

[19] http://www.netlib.org/lapack/

[20] http://www.wikipedia.org

VITA

Anurag Srivastava was born in Allahabad, north of India on June 1, 1981. He graduated in Information Technology from Indian Institute of Information Technology, Allahabad as one of the meritorious scholars in the batch, and joined WIPRO technologies as a systems engineer after college. As an undergrad student at IIIT, he researched in Pattern recognition and Machine learning. He published his work on neural networks in the "Conference on Thermal Systems, 2003" and his contribution to the fingerprint recognition at IIT-Kanpur in "Joint Conference on Information Sciences, 2003".

In 2004, he decided to pursue his interest in mathematical modeling and came to Virginia Tech to work with Dr Adrian Sandu. Being at Virginia Tech, he explored the area of atmospheric modeling and supercomputing applications. As his part-time job, he also developed parts of the gene-networks simulation software- Copasi under the supervision of Dr Pedro Mendes.