Converting Pollution Transport Code from HPF to OpenMP

Ioannis Tsilkos

August 17, 2011

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2011
Abstract

This research study has been performed in order for an air pollution application, that is called FRAME, to be converted in a Shared Memory parallel application. The FRAME model has been developed using Fortran and has been parallelised using a Data Parallel model by the Centre of Ecology and Hydrology (CEH). During this study, a research procedure has been carried out in order for the parallelisation methods that have been used on the FRAME model’s HPF version to be identified. Afterwards, an implementation procedure has been performed in order for this model to be converted into an OpenMP parallel application. This study concludes with the performance comparison between these two parallel versions. This research procedure suggests that the OpenMP version achieves better performance than the HPF version. However, due to the fact that HPF can be executed on more processors than OpenMP, these two parallel applications reach almost the same execution time.
## Contents

1 Introduction ................................................................. 1  
  1.1 Project Aims ......................................................... 1  
  1.2 Report Structure .................................................. 2  

2 The Background Knowledge and the Related Work ................ 4  
  2.1 The Air Pollution Background ..................................... 4  
  2.2 FRAME Model .......................................................... 6  
    2.2.1 FRAME Model’s Input Data .................................. 6  
    2.2.2 Distribution of the UK Domain ............................... 7  
    2.2.3 The Vertical Diffusion in each Air Column ............... 9  
    2.2.4 FRAME Model’s Output Data ................................ 10  
  2.3 HPF Implementation Background .................................. 11  
  2.4 The Related Work .................................................. 12  
  2.5 Hardware and Parallel Architecture ............................. 14  

3 Preliminary Findings ..................................................... 15  
  3.1 The three Version FRAME Model Implementation ............. 15  
    3.1.1 The Conditional Compilation ............................... 16  
    3.1.2 Compilation and Execution ................................ 17  
  3.2 The High Performance Fortran Parallel Programming Model ... 19  
    3.2.1 The HPF Execution in the FRAME Model ................. 19  
    3.2.2 The HPF_LOCAL Extrinsic Model .......................... 19  
    3.2.3 The Execution of the FRAME Model by the HPF_LOCAL .. 20  
    3.2.4 Input and Output arrays’ Distribution .................. 22  

4 The Implementation of the FRAME model’s OpenMP Version ...... 23  
  4.1 The OpenMP Parallel Programming Model ....................... 23  
    4.1.1 The OpenMP Execution .................................... 24  
  4.2 The Parallelisation Strategy .................................... 24  
  4.3 The Implementation of the OpenMP Version .................... 25  
    4.3.1 The Parallel Section ...................................... 25  
    4.3.2 The OpenMP Directives .................................... 25  
    4.3.3 Problems that have been Encountered .................... 27  
    4.3.4 Threadprivate Variables ................................. 28  
    4.3.5 The Communications among the Threads .................. 28
List of Tables

5.1 The Execution Time of the FRAME model’s Serial Sections ....... 32
A.1 The Execution Time Results of the FRAME Model’s Serial Version .. 50
A.2 The Execution Time Results of the HPF Version’s Performance Tests .. 50
A.3 The Execution Time Results of the different Schedules’ Performance Tests ............... 51
A.4 The Execution Time Results of the OpenMP Version’s Performance Tests 51
# List of Figures

2.1 The Transport of the Air Pollution. ....................................... 5
2.2 The Distribution of the UK Domain. ..................................... 7
2.3 The Different Executions of the FRAME Model. ...................... 8
2.4 The FRAME Model’s Vertical Diffusion. ............................... 9
2.5 The FRAME Model’s Output. ........................................... 10
2.6 The CRS4 Project’s Results. ........................................... 13

3.1 An Application that Includes both HPF and OpenMP Directives .... 16
3.2 A Conditional Compilation Example ................................... 17
3.3 A Makefile that Supports the Conditional Compilation ............ 18
3.4 The Declaration of a Subroutine as HPF_LOCAL .................... 20
3.5 The HPF_LOCAL Execution. ........................................... 21
3.6 The Array Distribution that has been Performed by the HPF ....... 22

4.1 An OpenMP Parallel DO loop ........................................ 26
4.2 The Threadprivate Declaration ....................................... 28
4.3 The Usage of the Critical Sections in FRAME Model ............... 29
4.4 The DATA Construct Usage in a Subroutine ......................... 30

5.1 The HPF Version’s Performance. ..................................... 33
5.2 The Static Schedule without Chunk. .................................. 35
5.3 The Static Schedule with Chunk Size Equals to 2 .................. 36
5.4 The Performance of the Static Schedule for 8 Threads and for a number of different Chunk Sizes. ................................. 36
5.5 The Distribution Strategy of the Dynamic Schedule with Chunk Equals to 2 .............................................................. 37
5.6 The Performance of the Dynamic Schedule for 8 Threads and for a number of different Chunk Sizes. ................................. 38
5.7 The Distribution Strategy of the Guided Schedule with Chunk Equals to 1 .............................................................. 39
5.8 The Performance of the Guided Schedule for 8 Threads and for a number of different Chunk Sizes. ................................. 39
5.9 The Performance Comparison among the Schedules. ............... 40
5.10 The OpenMP Version’s Performance. ............................... 42
5.11 The Performance Comparison between OpenMP and HPF ......... 43

B.1 The Ammonia Products’ Concentration and Deposition. .......... 53
B.2 The Nitrogen Products’ Concentration and Deposition. . . . . . . . . . . 54
B.3 The Nitrogen Products’ Concentration and Deposition. . . . . . . . . . . 55
B.4 The Sulphur Dioxide Products’ Concentration and Deposition. . . . . . 56
B.5 The Sulphur Dioxide Products’ Concentration and Deposition. . . . . . 57
Acknowledgements

I would like to thank Dr. Stephen Booth for his patience and his advice during this study. His contribution was crucial for the outcome of this research. I would also like to thank Dr. Tony Dore who helped me to adapt to the Centre of Ecology and Hydrology, and also for his advice on the meteorology background.
Chapter 1

Introduction

This technical review is the description of a research dissertation which has been implemented in terms of the MSc High Performance Computing of the University of Edinburgh. In this study, which has been performed on behalf of the Centre of Ecology and Hydrology (CEH), an air pollution model, which is called FRAME, has been used in order to be converted into a Shared Memory parallel application. The FRAME model has been developed using Fortran and has been parallelised using a Data Parallel model. After the implementation of the FRAME application’s Shared Memory version, a benchmarking process has been performed in order for the performance of these two parallel versions to be evaluated. Finally, a comparison between these two parallel programming models has been implemented.

1.1 Project Aims

The main aim of this study is to be developed a new parallel version of the FRAME model which is more portable than the Data Parallel which is already available. In order for this aim to be achieved, the OpenMP, which is a portable and also scalable Shared Memory parallel programming model, has been used [1]. In addition, the fact that the Data Parallel version of the FRAME model, which has been implemented using High Performance Fortran (HPF) [2], reaches very good execution time, has created an additional goal. This goal can be described by the investigation of the HPF methods that are used on the Data Parallel version, and how these methods affect the application’s execution time.
After the completion of this study, a new FRAME application, which includes three different versions of the FRAME model, should have been implemented. These three different versions are the serial, the Data Parallel implemented using High Performance Fortran and the Shared Memory implemented using OpenMP. The main goal of this implementation is to be supported the good execution time of the HPF version and the portability of the OpenMP version in the same application. Furthermore, the serial source code has also to be included in order for its future usage for further implementation to be supported. Finally, this new application should give the opportunity to the user to choose the version that will be executed by giving a parameter during the compilation time and by using the appropriate script file.

The last aim of this project is to be performed a benchmarking process in order for the performance evaluation of these two parallel versions of the FRAME model to be achieved. In addition, a performance comparison between the Data Parallel and the Shared Memory parallel programming models has also to be implemented. In order for the performance evaluation and the performance comparison of these two parallel programming versions to be achieved, the Nemesis [3], which is the High Performance system of the Centre of Ecology and Hydrology, has been used. Due to the Symmetric Multiprocessing architecture of this system, the two parallel versions of the FRAME model should be tested in a single node in order for their performance to be checked in a shared memory architecture [3].

1.2 Report Structure

During this study, in order for the previously mentioned aims to be achieved, a number of independent tasks have been addressed. This implementation suggests that this project is comprised of a number of different parts. This approach has also been used in order for this technical review to be implemented. More specifically, this study has been divided in the previously mentioned goals, which have been described in a number of chapters. These chapters include the description of the implementation methods that have been performed, and the presentation of this project’s findings. Below, there is a brief description of the following chapters.

The following chapter (Chapter 2), includes the background knowledge that has been used in order for this study to be implemented. More specifically, the transport of the air pollution that is simulated by the FRAME model is described. In addition, there is a detailed presentation of the methods that are performed by the serial version, in order for the previously mentioned simulation to be achieved. Furthermore, the reasons for the implementation of the FRAME model’s HPF version are discussed. Finally, a study with a relevant background is described and afterwards, the features of the High Performance system, that has been used during this study, are presented.
In the Chapter 3, the knowledge that has been gained before the beginning of the OpenMP version’s implementation procedure is described. More specifically, the methods that have been performed in order for the initial model to be converted in an application which includes the three different versions of the FRAME model, are presented. In addition, there is a detailed description of the distribution methods that are used by the HPF in order for the serial version of the FRAME model to be executed in parallel.

In the Chapter 4, the implementation methods that have been used in order for the FRAME model’s serial version to be converted into a Shared Memory parallel application, are presented. In addition, the problems that have been encountered during this implementation have been discussed. Finally, the testing procedure which has been performed in order for the correctness of this new application to be checked, is described.

In the Chapter 5, there is a detailed description of the performance tests that have been implemented. More specifically, the HPF version has been tested for a number of different processors and the results are discussed. In addition, a number of different distribution methods that are supported by OpenMP have been tested and the one which reaches the best possible performance has been identified. This chapter concludes with the performance measurement of the OpenMP version and with the comparison of these two parallel versions.

Finally, the Chapter 6, includes the conclusions and in the Chapter 7, a few ideas for the future implementation of this study are presented.
Chapter 2

The Background Knowledge and the Related Work

In this chapter, there is a presentation of the background knowledge that has been used for the implementation of this project. Initially, the transport of the air pollution, which is simulated by the FRAME model, is described. In addition, there is a detailed outline of the methods that are performed by the serial FRAME model, in order for the previously mentioned simulation to be achieved. Furthermore, the reasons for the development of the FRAME model’s HPF version are encountered. Finally, this chapter concludes with the findings of a study with a relevant background and the detailed description of the features of the parallel architecture that has been used, in order for the performance of these two parallel versions of the FRAME model to be measured.

2.1 The Air Pollution Background

In late 1960s, the air pollution phenomenon, in which a number of pollutants can be transposed over long distances between European countries and to damage ecosystems that are far away from the source of the emission, had caused ecological issues. During this period, this phenomenon was only a scientific hypothesis but a few years later, between 1972 and 1977, it was confirmed by several studies. For this ecological impact to be reduced, an international environmental law was passed in 1979 during the Genova Convention. By this law, the countries have been enforced to reduce the concentration of the pollutants that are emitted. The last decade, the United Kingdom (UK) has contributed at this global effort by the reduction of the emission of a few chemicals such as the sulphur dioxide (SO$_2$) and the nitrogen oxides (NO$_x$). Nevertheless, the emission of the ammonia (NH$_3$) has not the same decrease rate with the previous pollutants. [4]
The transport and the deposition of the air pollutants is a complex process which depends on a number of different parameters, such as the land cover and the rainfall rate of a specific area. In order for this transport to be simulated a number of models, which are called Atmospheric transport models, have been implemented. These models simulate "the transport, the chemical interactions and the deposition of these air pollutants and finally they predict the concentration and the deposition of them for the next period at a reasonable spatial scale" [4]. The FRAME model, which has been used in this project, is also an Atmospheric transport model which simulates the life cycle of a number of air pollutants such as the ammonia (NH$_3$), the sulphur dioxide (SO$_2$) and the nitrogen oxides (NO$_x$) which contribute to acid deposition and nitrogen deposition over the British isles. [4]

The air pollution phenomenon that is simulated by the FRAME model begins when the ammonia, the sulphur dioxide and the nitrogen oxides are emitted to the atmosphere. It is a fact that the source of emission is different for each pollutant. More specifically, the SO$_2$ is emitted by the combustion of fossil fuels in electricity generating power stations, industrial plants, residential heating, commercial and service sectors. Additionally, the NO$_x$ is emitted by the road transport, shipping and aircraft. Finally, the main source of emission for the ammonia is the agriculture activities such as the animal farming and the soil fertilisation. Below, the atmospheric phenomenon which is caused by these pollutants has been described, and has also been represented in the Figure 2.1. [4]

![Figure 2.1: The Transport of the Air Pollution.](image)
The atmospheric ammonia, is not an acid such as the SO$_2$ and the NO$_x$ but is highly reactive. This has as a result when it is emitted to the atmosphere to react with other chemicals and to be converted to ammonium aerosols (NH$_4^+$). This reaction also depends on the presence of other chemicals, which are products of the SO$_2$ and the NO$_x$. When these pollutants are formed, they can be transported away from their source of emission and they can also be deposited to the land by dry deposition or wet deposition (Figure 2.1). This phenomenon results in the acidification of soil and fresh waters. This can be described by the increment both of the pH balance and other nutrients’ concentration balance. It is a fact that these two factors can cause the damage of natural ecosystems. In addition, they have also negative impact on the earth’s climate and on the humans’ health due to the formation of particulate matter in the atmosphere. [4]

2.2 FRAME Model

As it was previously mentioned, the FRAME model is an Atmospheric transport model which estimates the environmental impact of the ammonia, the sulphur dioxide and the nitrogen oxides, over the United Kingdom (UK) [4]. This application has been developed using Fortran and has been parallelised using High Performance Fortran by the Centre of Ecology and Hydrology (CEH). In order for the FRAME model’s simulation methods to be investigated, below there is a detailed description of the serial version of this air pollution application.

2.2.1 FRAME Model’s Input Data

The FRAME model, at the beginning of its execution, reads a number of input data which can be divided in a number of meteorological data and the files which include the emission estimates of each pollutant. The meteorological data is used in order for a more realistic representation of the UK to be created. More specifically, these inputs are the wind speed and frequency, which determine the amount of time that is needed in order for a pollutant to be transposed from a specific area to another. In addition, the different land covers affect the rate of pollutants that are deposited in each ecosystem by the dry deposition. Finally, the rainfall rates affect the wet deposition of the pollutants. These meteorological data have been collected by a number of different sources [4]. As noted by Dr. Tony Dore, "the database of the wind data has been created using radiosonde data". The radiosonde is a device which is transferred to the upper atmosphere by a balloon [5]. During this transfer, a number of meteorological measurements, such as the humidity and the air pressure are collected [5]. The land cover data have been collected from the Land Cover maps of the Centre of Ecology and Hydrology and the rainfall rates have been estimated by the Meteorological Office. [4]
As far as the pollutants’ measurements concerned, the FRAME model reads a number of databases which include the concentration of the emissions for the ammonia, the sulphur dioxide and the nitrogen oxides for a specific year. These data have also been collected by a number of different sources [4]. The ammonia database has been created by Sutton and Dragosits and the databases of the sulphur dioxide and the nitrogen oxides emissions by the National Atmospheric Emissions Inventory. [4]

### 2.2.2 Distribution of the UK Domain

After the completion of the reading procedure of the input data, the FRAME model distributes the UK domain in a number of grids which have dimensions 172 x 244. Each grid cell has 5x5 km$^2$ resolution and represents an air column in which the vertical diffusion of the pollutants is simulated. The FRAME application, in order to decide which air column to execute, takes into account the different wind directions of the air. These wind directions are determined by the angle with which the wind is imported in the UK domain. Inside the UK domain, the wind is represented by straight lines, which are called trajectories, and they traverse a number of different air columns. Below, in the Figure 2.2 the distribution of the UK domain has been depicted. In order for the Figures 2.2 and 2.3 to be created, a few images of Massimo Vieno’s study have been used. These images have been improved by myself and the contributions of Dr. Dore Tony. [6]

![Figure 2.2: The Distribution of the UK Domain.](image)
The FRAME model, in order to execute this procedure, supports two different ways. In the first one, the application takes into account the 360 different wind directions and the simulation is executed for all the possible number of trajectories. The other execution is simpler and has shorter execution time. During this execution, the FRAME model chooses only one air column and executes the simulation only for the number of the trajectories that cross this air column. In the Figure 2.3 the two different executions of the serial FRAME model are presented. [6]

Figure 2.3: The Different Executions of the FRAME Model.

*a) The FRAME model takes into account all the wind directions. Here the 0° trajectory is represented. b) The FRAME model executes the simulation for the trajectories which cross a specific air column.*
2.2.3 The Vertical Diffusion in each Air Column

As it was previously mentioned, the FRAME application simulates the vertical diffusion of the pollutants in each air column. In order for this procedure to be implemented, each air column is represented by a 33 layered system which has 2500 m altitude from the ground. In addition, each layer has different thickness from the others. More specifically, the layer which is near the surface has resolution 1m, as opposed to the layer at the top of the air column which has resolution 100 m. The simulation begins with the emission of the pollutants in the specific air column. The FRAME application also includes a Plume Rise model for major point source emissions (i.e. power stations), in order for the emission height, the velocity and the temperature of the sulphur dioxide and the nitrogen oxides to be calculated. After this procedure, these pollutants react chemically with the gaseous form of ammonia and finally the new chemicals are deposited to the surface by the dry or the wet deposition. The wet deposition is the procedure in which the pollutants are deposited to the ground by the rainfall and on the contrary, the dry deposition is the procedure in which the pollutants are directly deposited to vegetation. Below, in the Figure 2.4, the previously mentioned execution has been presented. In order for this Figure to be created, an image of Massimo Vieno’s study has also been used but has been changed by myself and the contributions of Dr. Dore Tony. [7]
2.2.4 FRAME Model’s Output Data

After the completion of the vertical diffusion’s simulation for each air column, a number of partial results for each trajectory have been calculated [4]. Afterwards, these results are combined in order for a number of output files, which include the estimation of each pollutant’s annual concentration and deposition, to be created [4]. Finally, a software program, which is called IDL, is used in order for these output files to be converted in a number of maps which draw the previously mentioned results. In the Figure 2.5 a deposition map, that has been created by the IDL, has been presented. In this map there is a presentation of how the ammonia products which are deposited by dry deposition affect the ecosystems of the UK.

![Figure 2.5: The FRAME Model’s Output.](image)

*The affection of the UK by the ammonia products which are deposited by dry deposition.*
2.3 HPF Implementation Background

A few years before, after the implementation of the FRAME model’s serial version by the Centre of Ecology and Hydrology, the application’s execution time was about 8.5 days. Due to this fact, the FRAME model’s developers decided to optimise this application, in order for the best possible performance to be achieved. By this optimisation process, the source code of the FRAME application was re-arranged in three different well defined sections. These sections, which have been described below, are the start, the kernel and the exit. [4]

- **start**: In this section, the number of trajectories that are going to be executed is determined and also all the input files are read.
- **kernel**: In this section, the air pollution phenomenon is simulated for each trajectory and the partial results are stored in internal arrays.
- **exit**: In this section the trajectories’ partial results are combined and after the usage of a few statistical methods a number of output files which present the final results are created.

By the completion of the optimisation process, a profiling procedure revealed that the kernel was responsible for the application’s large execution time. This finding can also be explained by the fact that the serial FRAME model’s large execution time is due to the consecutively execution of the trajectories. In addition, it was also known that the application executes the same calculations for each trajectory and this has as a result the trajectories’ independent execution. This execution approach was convenient in order for a parallel version of the FRAME application to be developed. In order for this parallel version to be implemented, the High Performance Fortran (HPF) Data Parallel programming model was used. This parallel model was preferred to the Message Passing parallel model due to the ease of its implementation and due to the shared memory architecture of the available system. After the optimisation and the parallelisation procedure combined with access to a new High Performance system, the FRAME model’s execution time was reduced to a few minutes. In the previous studies there is not a detailed description of the methods that are used by the HPF in order to distribute the trajectories over the available processors, but under this project’s aims a research procedure has been implemented and the parallelisation methods of the HPF have been identified. [4]
2.4 The Related Work

During this study it was not able to be discovered any similar research on relevant subject except from a study of the CRS4, which is an Italian research center [8]. In this project, a scientific group of this research center, tried to convert an HPF geophysics application into OpenMP [9]. This application was a 3-D image processing programme which had been distributed using HPF over the third dimension, which is the image’s depth [9]. At each depth, the application executes a number of independent calculations for each frequency and finally a 2-D image is created by the contribution of each frequency [9]. The data parallel nature of this geophysics application is similar to the FRAME model, which uses a number of concurrent calculations for each trajectory.

In order for the performance of the HPF and the OpenMP parallel versions of the 3-D image processing application to be compared, the CRS4 center’s scientific group used a distributed shared memory machine which is called Origin2000. The performance tests that were implemented in this ccNUMA architecture, suggested that the HPF parallel version achieves better performance and better scalability than the OpenMP version (Figure 2.6). This performance difference was justified by the methods that are used by OpenMP in order to manipulate the memory. The fact that the memory of the Origin2000 is organised within pages, had as a result the distribution of the arrays over the available processors to be implemented by pages and not by elements. In order to be achieved this distribution, each memory page was located at the local memory of the first processor that read this page. However, this distribution method had as a result a number of sub-arrays to be located in different local memories and the memory access to be very expensive. On the other hand, the HPF parallel version applies data distribution on the arrays and each sub-array is located at the local memory of the processor which is going to work on it. It is obvious that the distribution method that is used by the HPF minimise the communications among the available processors and due to this fact achieves better performance than the OpenMP implementation. Below, in the Figure 2.6 the findings of the CRS4 project are presented. [9]
These results suggest that in the study of the CRS4 research center, the HPF version performs better than the OpenMP. [9]

The lack of similar studies, as well as the memory allocation problems that were encountered during the CRS4 center’s project, make the decision of the parallelisation strategy, which has been used in order for the FRAME model’s OpenMP version to be implemented, difficult. In order for this risk to be reduced, the distribution methods and the parallel sections of the FRAME model’s HPF version, have been identified. After the gain of this knowledge, the parallelisation strategy, which has been chosen for the development of the OpenMP version, has been focused on the parallelisation of a few sections that have already been parallelised by the HPF. In addition, a research procedure has been performed in order for the appropriate distribution strategy to be used and the best possible data locality to be achieved. In the next chapter, there is a detailed description of the methods that have been used by the HPF in order for the FRAME model to be executed in parallel.
2.5 Hardware and Parallel Architecture

As it was previously mentioned, in this study, the High Performance system of the Centre of Ecology and Hydrology, which is called Nemesis, has been used. In this system, the FRAME model’s OpenMP version has been developed and afterwards, a number of tests have been performed in order for the performance evaluation of the two parallel versions to be measured. The Nemesis, is a High Performance system which is based on Beowulf clusters [3]. More specifically, this system is consisted of 1 master and 26 slave nodes and each node includes 2 processors [3]. The Nemesis has the Symmetric Multiprocessing (SMP) cluster architecture. This multiprocessor computer hardware architecture can be described by a number of SMP nodes which are connected by an interconnect mechanism [10]. In addition, in each node there is a number of identical processors which can possibly share a global memory [10]. The Nemesis in order to set the communications among the nodes, uses a InfiniBand inter-process communication switch [3]. As far as the nodes’ architecture concerned, there are 2 processors (Intel quad running at 2.8Ghz) which support 4 threads each one and they share 32GB of RAM [3]. In addition, they communicate by a InfiniBand inter-process communication link [3]. In order for the performance of the FRAME model’s two parallel versions to be measured in a shared memory architecture, the performance tests have been implemented only in one node.
Chapter 3

Preliminary Findings

Before the beginning of the implementation of the FRAME model’s OpenMP parallel version, which is the main aim of this study, two more preliminary goals have to be achieved. The first one is the development of an application which includes the three different versions of the FRAME model and the other one is the identification of the parallelisation strategy that is used by HPF in order to execute the FRAME model in parallel. In this section, there is a detailed description of these two preliminary aims. Initially, the methods that have been used for the development of the new application have been described. Afterwards, the findings of a research procedure, which has been implemented in order for the HPF version’s parallelisation strategy to be identified, are discussed.

3.1 The three Version FRAME Model Implementation

At the beginning of this project, the development of an application which can include the serial, the HPF and the OpenMP versions of the FRAME model, was just an assumption. In order for the achievement of this aim to be confirmed, a few problems have been solved. Initially, a few methods which can distinguish the different versions of the FRAME model have been discovered. In addition, an implementation strategy for the development of a Makefile which includes the different compiler options has been identified. Finally, a script file for the execution of each version has been created. In this section, there is a detailed description of the implementation methods that have been used in order for this new application to be developed.
3.1.1 The Conditional Compilation

The parallelisation of a serial application by the HPF or the OpenMP parallel programming models can be achieved by only a few additional lines of source code [11]. These pieces of code determine the sections that have to be executed in parallel and also specify the way with which the available processors have to cooperate in order for the best possible performance to be achieved [11]. Following this observation, in order for the new FRAME application to be implemented, a few methods which introduce the conditional compilation of a few pieces of source code have been used. The fact that HPF and OpenMP support the !HPF$ and the !$OMP directives respectively, which can be read as comments from the serial compiler, is a convenient approach in order for the parallel versions to be distinguished from the serial [12][13]. In addition, this distinguish is also applicable between the OpenMP and the HPF parallel versions. Below, there is an example which represents how the different directives can be included in the same programme.

**Program 1** A Programme which Includes HPF and OpenMP Source Code

```
PROGRAM life

!$ USE OMP_LIB

!HPF$ DISTRIBUTE(CYCLIC,CYCLIC) :: board,tmpboard,neighbours
INTEGER , DIMENSION(100,100) :: board,tmpboard,neighbours
INTEGER :: maxloop,loop=0
........
!$OMP PARALLEL PRIVATE(loop) SHARED(maxloop)
!$OMP DO SCHEDULE(STATIC, 4)
       DO loop=1,maxloop
           ........
       END DO
!$OMP END DO
!$OMP END PARALLEL

END PROGRAM life
```

Figure 3.1: An Application that Includes both HPF and OpenMP Directives
However, in order for the conditional compilation to be achieved, except from the HPF and the OpenMP directives, a few other Fortran sections, such as the initialisation of a specific variable, have to be distinguished. This conditional compilation, can be achieved by the usage of a compiler flag, which is called -D [14]. By this compiler flag, a name, with which the compiler can identify the sections that has to take into account during the compilation time, is declared [14]. In the source code, this name has to be used in conjunction with the #ifdef and #endif directives, in order for the conditional compilation of these sections to be explicitly declared [15]. This method has been used for the distinguish of the serial and the HPF version. The distinguish of the OpenMP version, has been achieved by the !$ directive which reaches similar execution with the -D compiler flag [16]. Below, there is a presentation of a piece of the FRAME source code which use these conditional compilation methods.

Program 2  An Example of the Conditional Compilation

```fortran
#ifdef serial
    INTEGER,PARAMETER :: nprocs= 1
#endif

#ifdef hpf
    INTEGER,PARAMETER :: nprocs= 8
#endif

!$ INTEGER,PARAMETER :: nprocs= 1
```

Figure 3.2: A Conditional Compilation Example

3.1.2  Compilation and Execution

After the necessary changes that have been performed in the FRAME model’s source code, a Makefile and a few script files suitable to this application have been implemented. In order to be compiled the several versions of the FRAME application, a number of different compilers which use different compiler flags have been used. More specifically, the serial version has been compiled with the pgf90 compiler, the OpenMP version with the same compiler in conjunction with the -mp compiler flag and the HPF with the pghpf compiler [14]. The Makefile that has been implemented, in order to simplify the compilation procedure of the FRAME model, includes the appropriate compiler as well as its compiler flags for each version. The choice of the version that is going to be compiled, can be determined by a parameter that is given by the user during the compilation time. As regards the execution of this new application on the back-end
of the Nemesis, a few script files have been used. The script files for the execution of
the serial and the HPF version of the FRAME application were already implemented.
However, there was not any script file for the execution of an OpenMP application,
but under this project’s aims a suitable script file has been created. Below, there is an
example of a Makefile that supports the conditional compilation, and also includes the
different options for each FRAME model’s version.

Program 3 An Example of a Makefile that Supports the Conditional Compilation

SERIAL_FC=pgf90
HPF_FC=pghpf
OPENMP_FC=pgf90

SERIAL_OPT= -Dserial=1
HPF_OPT= -Dhpf
OPENMP_OPT= -mp

SERIAL_FFLAGS=$(SERIAL_OPT)
HPF_FFLAGS=$(HPF_OPT)
OPENMP_FFLAGS=$(OPENMP_OPT)

all: serial hpf omp

serial: $(SRC)
   $(SERIAL_FC) $(SERIAL_FFLAGS) $(SRC) -o serial.exe

hpf: $(SRC)
   $(HPF_FC) $(HPF_OPT) $(SRC) -o hpf.exe

omp: $(SRC)
   $(OPENMP_FC) $(OPENMP_FFLAGS) $(SRC) -o omp.exe

Figure 3.3: A Makefile that Supports the Conditional Compilation
3.2 The High Performance Fortran Parallel Programming Model

The High Performance Fortran (HPF), which is a data parallel programming model, was initially published by the High Performance Fortran Forum (HPFF) in 1993 [17]. The HPF was implemented in order for the Fortran programming language to be extended by a few directives, which provide access to high performance architecture features [17]. Using this parallel programming model, a serial Fortran application can be executed in parallel by the distribution of the arrays, that need large computational power in order to be executed, over the available processors [18]. The involvement of the HPF directives in this array distribution is to inform and to provide assertions to the compiler which is responsible for this distribution [18]. It is obvious that the performance of a parallel Fortran application which has been implemented using HPF, depends on the methods that the compiler uses in order to distribute the data over the available processors and to optimise the application [19]. As it was previously mentioned, the HPF parallel programming model has been used in order for the FRAME application to be executed in parallel. Below, there is a detailed description of the parallelisation strategies that have been performed by the HPF compiler, in order for this parallel execution to be achieved.

3.2.1 The HPF Execution in the FRAME Model

The HPF parallel programming model has used a data distribution method in order to convert the FRAME model in a parallel application. This parallelisation has been applied in a few subroutines by the usage of the extrinsic HPF_LOCAL model. This model applies the multi-threaded execution in an HPF application [20]. In order for this execution to be achieved, the input arrays, that store the input data, and the output arrays, that store the partial results of each trajectory, have to be distributed over the available processes [20]. In order for the compiler to be informed for these arrays’ distribution the HPF directives have been used. Below, there is a detailed description of the parallelisation method that has been used.

3.2.2 The HPF_LOCAL Extrinsic Model

Generally, the execution method that is used by an HPF programme differs from the execution of other parallel programming models such as the MPI. In order for an HPF programme to be executed, a single-threaded "HPF Global" model is used. In this context, the thread terminology is used to mean a logic thread of control in the programme rather than an execution thread. In this model, only one thread is responsible for the execution of this programme, regardless of the number of the processors that execute it in parallel. During this single-threaded execution, only the sections that include data parallel statements and HPF directives can be executed in parallel. The parallel execution
of these sections is achieved by the usage of as many possible processors as are available and varies depending on the distribution methods that have been chosen. However, it is also possible for an HPF application to achieve better performance, if a few sections of this application can be executed by the multi-threaded “local” execution model, which is supported by other parallel programming models such as the MPI. In order for an HPF application to achieve this multi-threaded execution, the HPF_LOCAL directive is supported by the HPF parallel programming model [20].

3.2.3 The Execution of the FRAME Model by the HPF_LOCAL

In the HPF version of the FRAME application, a large number of the external subroutines have been declared as extrinsic HPF_LOCAL (Figure 3.4). By the declaration of these subroutine as HPF_LOCAL, the PGHPF compiler is informed that these sections have to be executed by the multi-threaded execution [20]. In order for this execution to be achieved, each processor executes these subroutines only for the data that are stored in its own local memory [20]. In the Figure 3.4 there is an example in which a subroutine is declared as HPF_LOCAL. Before the usage of this subroutine, an interface has to be used in order for the compiler to be informed for this declaration [20].

Program 4 The Usage of the HPF_LOCAL

INTERFACE
EXTRINSIC (HPF_LOCAL) SUBROUTINE frame
  USE mainpar
  USE inpvar
  USE modvar
END SUBROUTINE frame
END INTERFACE

EXTRINSIC (HPF_LOCAL) SUBROUTINE frame
  USE mainpar
  USE inpvar
  USE modvar

  ................

END SUBROUTINE frame

Figure 3.4: The Declaration of a Subroutine as HPF_LOCAL
During the declaration of the input and the output arrays, a variable, which stores the number of the processors that execute the application in parallel, is taken into account. This variable determines the length of these arrays’ last dimension. By the usage of the HPF_LOCAL, the input and the output arrays are distributed over this dimension, and each processor stores in its own local memory, only a part of the previously mentioned arrays [20]. Afterwards, each processor uses the single-threaded "HPF Global" execution model in order to execute the instruction set only for its own data [20]. In order for this execution to be achieved, each processor has also a copy of the instruction set in its own local memory [20]. By the usage of this method, a few subroutines and the large number of the trajectories are performed in parallel. Due to the fact that each trajectory is executed independently from the others, the communications that have to be set among the processors are the minimum possible. It is obvious that this execution approach is convenient for the usage of the multi-threaded execution model. As it was previously mentioned, this execution is similar to an MPI execution in which the data is distributed over the available processors, and each processor works on its own data [21]. Below, in the Figure 3.5, the execution that is achieved by the HPF_LOCAL directive is presented.

![Figure 3.5: The HPF_LOCAL Execution.](image-url)
3.2.4 Input and Output arrays’ Distribution

By the usage of the HPF_LOCAL directive, a few subroutines have been executed in parallel and the input and the output arrays have been distributed over the available processors. However, in order for this execution to be achieved, the arrays’ distribution strategy has to be explicitly declared [20]. In the FRAME model, this declaration has been achieved by the creation of an arrangement of abstract processors and afterwards, the assignment of each array to this arrangement [22]. This distribution has been performed by the usage of two directives which are the PROCESSORS and the DISTRIBUTE (Figure 3.6) [22]. The first directive informs the compiler how to arrange the available processors and the other one which distribution method to use, in order to map the arrays to the previously mentioned arrangement [22]. In the Figure 3.6, there is an example of the implementation that it was previously mentioned.

Program 5 The HPF Array Distribution

```c
#ifdef hpf
   INTEGER,PARAMETER :: nprocs= 8
#endif

!HPF$ PROCESSORS procs(nprocs)

!HPF$ DISTRIBUTE(CYCLIC) ONTO procs :: anglei
```

Figure 3.6: The Array Distribution that has been Performed by the HPF

The distribution method that has been used is the cyclic. This method informs the compiler that each array’s element has to be assigned to each processor consecutively [22].
Chapter 4

The Implementation of the FRAME model’s OpenMP Version

As it was previously mentioned, the main aim of this study is to convert the serial FRAME model in an OpenMP parallel application, and afterwards to achieve this application’s best possible performance. In order for this goal to be reached, a few problems have been solved before and during the development procedure. Initially, a research on the serial version’s source code has been implemented, in order for the sections that have to be executed in parallel to be identified. This research procedure has been performed carefully due to the fact that there is a large number of parallelisation strategies that can be used, but the one that reaches the best possible performance, should be chosen. In addition, the section that has to be executed in parallel by OpenMP, includes a large piece of source code in which a large number of external subroutines are called. It is a fact that this source code cannot be easily converted into a parallel section. As a result, an extra implementation effort has been required, in order for this parallelisation to be performed. Finally, during the implementation procedure, a few other problems have been encountered and have been solved. In this chapter, there is a detailed presentation of the development methods that have been used and the problems that have been solved in order for the FRAME model’s OpenMP version to be developed.

4.1 The OpenMP Parallel Programming Model

The Open Multi-Processing (OpenMP) parallel programming model, is a portable and also scalable model, which consists of a number of compiler directives. This parallel programming model is supported by a few programming languages such as C, C++ and Fortran, and is used in order for the development of a parallel application, which can be executed in Shared Memory architectures, to be achieved. [1]
4.1.1 The OpenMP Execution

The execution of the OpenMP applications is based on the notion of threads [23]. This multi-threaded execution can be achieved, by the usage of a fork-join model, which performs work distribution over the available threads [24]. The execution of an OpenMP application begins by the master thread which executes the application’s instruction set consecutively, until the beginning of a parallel section [24]. In order for this parallel section to be executed, the master thread forks a number of slave threads, and afterwards applies the work distribution over them [24]. After the conclusion of each thread’s execution, the master thread kills the slave threads and continues the consecutive execution until the next parallel section [24]. It is obvious that the OpenMP execution can be achieved by the usage of a pool of threads which can be used only for the parallel sections’ execution [24].

4.2 The Parallelisation Strategy

As it was previously mentioned, a research procedure has been performed, in order for the sections that have to be executed in parallel by OpenMP to be identified. It is a fact that the knowledge, which has been gained by the study of the FRAME model’s HPF execution, has also been used during this research procedure. After the identification of the serial FRAME model’s sections which have been parallelised by HPF, this procedure has been focused on these sections in order to discover which are more suitable to be parallelised by OpenMP. In addition, a profiling tool has also been used in order to measure the execution time of each section and to identify which of them need the most computational power in order to be executed. By the conclusion of this research, it is obvious that only the section, which takes into account the large number of trajectories, is responsible for the large execution time of the serial FRAME model. This section executes independently the simulations for all the available trajectories and simulates in each one, the air columns’ vertical diffusion. This execution approach is convenient for the parallelisation of this section. This can be explained by the fact that the minimum possible communications among the threads are needed, in order for this section to be executed in parallel.
4.3 The Implementation of the OpenMP Version

In this section, there is a detailed description of the development process that has been performed, in order for the FRAME model’s OpenMP version to be implemented. More specifically, there is a description of the piece of the serial source code which has been converted in an OpenMP section. In addition, the OpenMP directives that have been used are presented. Afterwards, the problems that have been encountered are discussed, and finally, there is a description of the communications that have been set among the threads.

4.3.1 The Parallel Section

As it was previously mentioned, in order for the air pollution phenomenon to be simulated, a number of smaller simulations have been performed for each trajectory. After the execution of each trajectory, a number of partial results have been stored in interval arrays and afterwards, they are combined in order for the final estimations to be calculated. The section that has been chosen in order to be converted into an OpenMP parallel section, includes a DO loop which executes consecutively each trajectory’s simulation. In order for these simulations to be performed, a large piece of sequential Fortran source code (1200 lines) has been developed and a large number of external subroutines have been called inside the parallel DO loop. The large number of variables that are included in this section, resulted in complex parallelisation task. However, in order for the FRAME model to simulate the previous air pollution phenomenon takes into account 107,760 trajectories. This large number of trajectories affects the number of the iterations that are executed by this section’s DO loop, and enforces its parallelisation. Finally, the independent execution of each iteration, is a convenient approach for this section’s parallelisation, due to the fact that the minimum possible communications among the threads are required.

4.3.2 The OpenMP Directives

After the identification of the section that has to be executed in parallel, in order for the FRAME model’s OpenMP version to be implemented, a development procedure has been performed. During this procedure, a few OpenMP directives have been used in order to convert the serial DO loop which was previously described, in an OpenMP parallel DO loop. In order for this implementation to be achieved, an OpenMP parallel section has to be declared out of the DO loop and afterwards a distribution method has to be used [25]. More specifically, the declaration of the parallel section can be achieved by the usage of the !$OMP PARALLEL and the !$OMP END PARALLEL directives [25]. These two directives, declare explicitly that the source code that is included has to be executed in parallel [25]. In addition, the nature of the variables that are used inside this parallel section have also to be declared. It is a fact that a variable can have different

25
values in each thread [25]. Under this situation, a copy of this variable is stored at the local memory of each thread and in this case is called local variable and is declared by the PRIVATE directive [25]. On the other hand, there are also variables which have to be available from all threads. These variables, which are called shared variables, are declared by the SHARED directive [25].

As far as the data distribution concerned, OpenMP supports a number of distribution methods which are called schedules. The schedules that are supported by OpenMP are the static, the dynamic and the guided. In order for these schedules to be performed, an OpenMP directive, which is called SCHEDULE is used. This directive accepts two parameters, the type which is the distribution method that has been used, and the chunk which is a parameter that declares the size of work that is assigned to each thread. Below, in the Figure 4.1, there is an example which represents how the OpenMP directives are used in order for a DO loop to be executed in parallel. [25]

```
Program 6 An OpenMP Parallel DO loop with the Static Schedule and Chunk Size 4

INTEGER :: maxloop, loop = 0
          ..........
!$OMP PARALLEL PRIVATE(loop) SHARED(maxloop)
!$OMP DO SCHEDULE(STATIC, 4)
   DO loop = 1, maxloop
       ........
   END DO
!$OMP END DO
!$OMP END PARALLEL
```

Figure 4.1: An OpenMP Parallel DO loop

During the implementation procedure of the FRAME model’s OpenMP version, the dynamic schedule has been used with the chunk parameter equals to 1. This schedule has been used indicatively, in order for the implementation of the FRAME model’s OpenMP version to be achieved, but afterwards a few tests have been implemented in order for the one which achieves the best possible performance to be identified. A detailed presentation of these performance tests is included in the next chapter.
4.3.3 Problems that have been Encountered

As it was previously mentioned, in order for the parallelisation of the DO loop to be achieved, except from the declaration of the OpenMP parallel section and the usage of the dynamic schedule, the variables that are used inside this loop, have to be declared as private or shared. In order for this declaration to be implemented, a research procedure for the identification of each variable’s usage has been performed. More specifically, the variables which are updated inside this loop have been declared as private and these which are only read, as shared. It is a fact that these two rules can not be applied in the parallelisation of each section, but for this DO loop can be used due to the independent execution of the iterations. During the implementation that it was previously described a number of problems have been encountered.

Global Variables’ Issue

The first problem that has been encountered, regards a few variables, which are updated in the external subroutines that are called inside the DO loop. These variables, which have been declared in a number of different modules, loose the updated value after the execution of the external subroutine. In order for this problem to be solved, an attempt has been made, in which these global variables have been passed as dummy arguments to the callable external subroutines and they have also been declared as private. However, this attempt has created a large number of issues and finally, has been abandoned.

The new Implementation’s Issues

As it was previously mentioned, during the attempt that was made in order for the global variables’ problem to be solved, a number of issues were created. Initially, a large effort was performed, in order for the external subroutines which use these global variables, to be identified and to be changed. In addition, the fact that the serial source code was changed, resulted in difficulties with this application’s further implementation. This can be explained by the fact that the initial developers have to spend time in order to understand and to identify the sections that have been changed. Finally, this new application tends to be OpenMP specific, and is also possible to affect the HPF version’s performance. Due to these issues, this approach was abandoned and a new implementation which has solved the initial problem was performed.
4.3.4 Threadprivate Variables

In order for the previous problem to be solved, these global variables have been declared as threadprivate. This clause is supported by OpenMP, in order for this parallel programming model to take into account the global variables which are specific for each thread [25]. This can be explained by the fact that each thread is able to store and to update a copy of a global variable, but this copy can also be accessed by all the other threads [25]. In order for the global variables that were previously mentioned to be declared as threadprivate, the THREADPRIVATE clause has been used [25]. It is worth mentioning that this declaration has not been implemented in the parallel section, but in the MODULES in which these variables were initially declared. Below, in the Figure 4.2, there is an example for the previously mentioned declaration.

Program 7 An Example of a MODULE which Includes THREADPRIVATE Variables

```
MODULE modvar
  REAL :: tStep
  REAL :: hnext
  REAL :: coszen
  !$OMP THREADPRIVATE(tStep,hnext,coszen)
END MODULE modvar
```

Figure 4.2: The Threadprivate Declaration

4.3.5 The Communications among the Threads

For the parallelisation of the previously mentioned DO loop to be concluded, a few communications among the threads have to be set. Due to the independent execution of the iterations, the communications that are needed have been minimised. However, a few called external subroutines, and the sections that perform the writing of the partial results to the output arrays have to be executed sequentially. In order for this sequential execution to be achieved, these sections have been declared as CRITICAL [26]. This directive is supported by OpenMP, in order to specify that the region of source code that is included, has to be executed by only one thread at a time [26]. Initially, as far as the called external subroutines which have to be executed in a critical section are concerned, they have to be executed sequentially, due to the fact that they include a local
array that is initialised by the DATA statement. This DATA statement sets the previously mentioned local arrays as shared [26]. This has as a result, each thread’s copy of these local arrays not to be updated only by the thread that owns them, but also from all the other threads that execute these called external subroutines. In order for this problem to be solved, the called external subroutines that include the previously mentioned arrays have been executed in a critical section. On the other hand, the sections in which the update of the output arrays is performed, have to be executed sequentially, in order for the correct order of the threads, which update these output arrays, to be set.

**Program 8** The Usage of the Critical Sections in FRAME Model’s OpenMP Version

```plaintext
EXTRINSIC(HPF_LOCAL) SUBROUTINE update

IMPLICIT NONE

REAL, DIMENSION(:, :) , INTENT(INOUT) :: oxDry
INTEGER :: nMStep, nWndtrj

INTERFACE
    EXTRINSIC(HPF_LOCAL) SUBROUTINE calMix
USE modvar
END SUBROUTINE calMIX
END INTERFACE

......

!$OMP PARALLEL SHARED( oxDry , nWndtrj ) PRIVATE(nMStep)
!$OMP DO SCHEDULE(DYNAMIC, 1)
    DO nMStep=1, nWndtrj
        .........
        !$OMP CRITICAL( mix )
        call calMix
        !$OMP END CRITICAL( mix )
        .........
        !$OMP CRITICAL( output )
        oxDry=oxDry+1
        !$OMP END CRITICAL( output )
    END DO
!$OMP END DO
!$OMP END PARALLEL
END SUBROUTINE update
```

Figure 4.3: The Usage of the Critical Sections in FRAME Model
Program 9 The calMix Subroutine that is called in a Critical Section from the update

EXTRINSIC (HPF_LOCAL) SUBROUTINE calMix

USE modvar
IMPLICIT NONE
REAL c l d F a c ( 0 : 2 )

DATA c l d F a c / 1 . 0 0 , 0 . 8 9 , 0 . 8 1 /

END SUBROUTINE calMix

Figure 4.4: The DATA Construct Usage in a Subroutine

4.4 Testing

During a software development procedure, a number of tests have to be performed, in order for the correctness of the application to be checked. It is also a customary to be used the Unit testing and the Integration testing. The Unit testing is used in order for the correctness of small units of source code to be checked, and on the other hand, the Integration Testing checks if a large number of individual software modules integrate properly [27]. Unfortunately, due to the structure of the DO loop which has been parallelised, the previously mentioned tests could not be used. However, a few different tests have been performed in order for the correctness of the FRAME model’s OpenMP version to be checked. Initially, during the implementation procedure, the variables that are used inside the DO loop have been printed on the screen and their values have been compared with the serial version’s. More specifically, the shared variables, which are only read inside the parallel section, have been printed and have been compared before the beginning of the parallel section. As far as the private and the threadprivate variables concerned, they have been printed and they have been compared in a number of different DO loop’s intervals. In case the values of the OpenMP version’s variables were not identical with the serial version’s, the possible errors were reported and the solution was implemented. After this iterative procedure, the serial version’s final results were reached and the first testing procedure was concluded. After the development of the FRAME model’s OpenMP version, a few other tests were performed, in order for the correctness of the application to be checked in a number of different threads.
Chapter 5

Performance Analysis

By the completion of the OpenMP version’s development, a new application which includes the three different versions of the FRAME model has been available. As it was previously mentioned, this application has also been used as a benchmark in order for the performance of the two parallel versions to be measured and then, the Data Parallel and the Shared Memory parallel programming models to be compared. In this section, there is a description of the tests that have been implemented and afterwards, a detailed analysis of their results.

More specifically, this chapter begins with the presentation of an experiment in which the serial version has been divided in a number of sections and the execution time of each one has been measured. This test has been performed in order to be identified how the serial sections of the FRAME model affect the whole application’s execution time. In addition, the HPF version’s performance is presented and the reasons for this performance are analysed. Furthermore, the OpenMP version has been tested for the different OpenMP schedules and the one which reaches the best possible execution time has been identified. Finally, the OpenMP application’s performance is discussed and afterwards, a comparison between these two parallel programming models has been performed.

5.1 The Serial Source Code’s Execution Time

According to the Amdahl’s law, "the potential program speedup is defined by the fraction of code (P) that can be parallelized" [28]. In case none of the code can be parallelised, the P is equal to 0 and the speed-up is equal to 1 [28]. In the other case, all the code can be parallelised and the P is equal to 1 and the speed-up infinite [28]. Following this argument, the FRAME model’s serial version has been tested, in order for the execution time of the sections, which cannot be executed in parallel, to be measured. In order for this aim to be achieved, a previously mentioned division of the FRAME model has been used. More specifically, the sections that have been measured are the
start, the kernel and the exit. It is already known that the start and the exit are executed sequentially and the kernel can be parallelised. This test suggests that the start and the exit need only a few seconds for their execution, and that the kernel is responsible for the application’s large execution time. More specifically, the execution time of each one of these two serial sections is about 4.5 seconds, while the kernel needs 10,764 seconds (179 minutes). It is obvious that the execution of the two serial sections does not affect the whole application’s execution time, which can be reduced significantly, by the usage of the appropriate parallelisation strategy. Below, in the Table 5.1 there is a detailed presentation of the execution time that has been reached by each section.

<table>
<thead>
<tr>
<th>Section</th>
<th>Seconds</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>4.451</td>
<td>0.074</td>
</tr>
<tr>
<td>Kernel</td>
<td>10,764</td>
<td>179.4</td>
</tr>
<tr>
<td>Exit</td>
<td>4.769</td>
<td>0.079</td>
</tr>
</tbody>
</table>

Table 5.1: The Execution Time of the FRAME model’s Serial Sections

5.2 The Performance of the HPF Version

During the experimental procedure in which the performance of the HPF version has been measured, a few tests have been implemented on the Nemesis. As it was previously mentioned, only one node of this parallel system has been used, in order for this FRAME model’s version to be checked on a shared memory architecture. During this procedure, the HPF version has been tested for a few different number of processors, and afterwards, the execution time of each one of these options has been measured. Due to the fact that these tests have been performed only on one node of the Nemesis, this FRAME model’s parallel version has been tested up to 8 processors.

In addition, in order for this version’s scalability to be evaluated, a performance measurement, that is called speed-up, has been used. The speed-up, in parallel computing, refers to how much better an application performs on a parallel machine, compared to the same application which is executed on only one processor [29]. This measure is also equal to the serial version’s execution time divided by the parallel version’s [29]. Finally, in order for the serial version’s execution time to be measured, the HPF version has been executed on one processor.
5.2.1 Performance Analysis

By the completion of the experimental procedure, that it was previously described, a graph, which represents the scalability and the performance of the HPF version, has been created. More specifically, this graph represents how the increment of the number of the processors affects the whole application’s speed-up. It is a fact that the HPF version achieves good enough execution time, about 27 minutes, when it is executed on 16 processors. However, the tests that have been implemented, suggest that this parallel version does not achieve the best possible performance and it does not scale very well. This finding can also be observed at the graph of the Figure 5.1, which has been presented below.

![HPF Performance Graph](image)

**Figure 5.1: The HPF Version’s Performance.**

According to this graph, the FRAME model’s HPF version does not scale very well. More specifically, the speed-up that is achieved when this parallel version is executed on two processors is almost equal to the serial speed-up (1.05). However, when this parallel version is executed on more than 2 processors the speed-up increases, but is still low compared to the linear speed-up. This performance can be explained by the source code’s sections that have been chosen in order to be executed in parallel by the HPF. In the next chapter, a few reasons which explain the HPF version’s low performance are discussed.
5.2.2 The Reasons of the HPF Version’s Performance

By the understanding of the HPF version’s parallelisation strategy, the sections which are executed in parallel have also been identified. However, a few of these sections do not need large time for their execution. This finding has been achieved during the profiling procedure that has been performed before the development of the FRAME model’s OpenMP version. It is obvious that this finding is also the main reason of this parallel version’s low performance. More specifically, in the HPF version, a large number of subroutines have been declared as HPF_LOCAL, but only a few of them are used inside the section in which the simulation is performed. In addition the input and the output arrays have been executed in parallel. However, the input arrays do not need large computational power for their execution. This can be proven by the fact that they are initialised in the start section which is executed in 4.5 seconds, and they are used inside the DO loop, in which only one value of them is used per iteration. This execution results in the application spending more time for the execution of the previously mentioned sections than the serial does. This can be explained by the fact that the application needs more time to set the communications among the processors, than to execute this specific section. Finally, there has also been an assumption that the ineffective memory allocation that may be achieved by the HPF version, contributes to the low performance. However, a few tests have been performed for a number of processors that are located in different nodes and the performance has been unchanged. Due to this fact, this assumption has been dispelled.

5.3 The Performance of the OpenMP Version

As it was previously mentioned, in order for the FRAME model’s serial version to be converted in a shared memory parallel application, a DO loop has been parallelised. This parallelisation has been achieved by the declaration of a parallel section out of the DO loop, and the usage of a distribution strategy which assigns each loop’s iteration to a specific thread. In addition, due to the fact that the OpenMP includes a few different distribution methods, that are called schedules, an experimental procedure has been performed in order to be identified which one reaches the best possible performance. During this procedure, this application has been tested on 8 threads for the different schedules and for a number of different chunk sizes. Afterwards, the speed-up of these tests has been calculated and the schedule which reaches the best speed-up has been chosen. In this section, there is a description of the different schedules and also a presentation of the performance that is achieved by each one.
5.3.1 The OpenMP Schedules

In an OpenMP parallel application, in order for the best possible performance to be achieved, the threads that execute this application in parallel, have to be load balanced [28]. This approach can be achieved when the threads are kept equally busy, and when the task idle time has been minimised [28]. It is a fact that the simplest distribution which can be performed, is the assignment of an equal number of iterations to each thread [25]. However, this method cannot perform well for all the applications, due to the fact that each iteration may need different computational power in order to be executed [25]. Following this observation, the best possible performance can be achieved by the usage of a distribution method which assigns the same amount of work to each thread [28]. As it was previously mentioned, in order for OpenMP to take into account the different executions that can be performed in a number of DO loops, supports a number of different distribution methods that are called schedules [25]. Below, there is a presentation of the parallelisation strategy that is used by each schedule, and how these different schedules affect the OpenMP version’s performance.

The STATIC Schedule

The STATIC schedule, can be used in an OpenMP application by two different ways. In the first one, which is the default, the chunk is undefined and the data distribution is achieved by the division of the iteration space in a number of pieces which are as many as the threads which execute the application in parallel (Figure 5.2). In addition, the size of all these pieces is approximately equal. After the conclusion of this procedure, each piece is assigned to each thread, in order for its execution to be performed. In the other one, the chunk is declared and specifies the size of each piece that is created after the division procedure (Figure 5.3). In case the iteration space is not divisible by the chunk, the last piece includes the number of the remaining iterations. After this procedure, the pieces are assigned to the threads in a "round-robin" fashion. In order for the Figures 5.2 and 5.3 to be created, two similar Figures of the Shared Memory Programming course slides have been used, but they has been improved by myself [25].
In order for the performance of the static schedule to be calculated, a number of tests have been performed for different chunk sizes. It is a fact that the default execution of the static schedule has not been tested. However, this execution is a special case of the static schedule with chunk. More specifically, when the chunk size is big enough in order to divide the iteration space in equal pieces, the default execution of this schedule is achieved. Below, in the Figure 5.4, a graph which includes the speed-up for the different chunk size options is presented.
The graph that is presented in the Figure 5.4 suggests that the static schedule achieves good enough performance. More specifically, the best performance (about 6.5 on 8 processors) is achieved, when the chunk size is smaller than 10. Afterwards, the performance gradually decreases as the chunk size tends to be equal to 100 and the speed-up reaches the 6.25. In addition, this schedule’s performance for chunk size between 100 and 1500 increases, and the speed-up achieves the 6.35. Finally, when the chunk size is over 1500 the performance varies in a range of values which are less than 6. It is worth mentioning that the speed-up is very low when the static schedule tends to achieve its default execution.

The DYNAMIC Schedule

The *DYNAMIC* schedule, uses a similar approach with the static, when this is executed with a chunk parameter, in order for the iteration space to be divided in a number of pieces. More specifically, the iteration space is divided in a few pieces of work which have size equal to the chunk. However, the dynamic method uses a different procedure in order for the distribution of these pieces to be achieved. These pieces are assigned consecutively to the first thread which finishes with the execution of the previous piece of work. Below, in the Figure 5.5 the distribution strategy that is used by the dynamic schedule is depicted. In order for the Figures 5.5 and 5.7 to be created, two Figures of the Shared Memory Programming course slides have been used, but they have been improved by myself. [25]

![Figure 5.5: The Distribution Strategy of the Dynamic Schedule with Chunk Equals to 2](image)

*The iteration space is divided in a number of pieces of work which have size equal to the chunk size. Afterwards these pieces of work are assigned to the first thread which is available.*

In order for the performance of this schedule to be checked, a testing procedure has been implemented. During this procedure, the dynamic schedule has been tested for
a different number of chunk parameters. The chunk size options that have been used during this procedure are these which have also been used during the static schedule’s performance tests. By the completion of this process, the graph which is presented in the Figure 5.6 has been created.

Figure 5.6: The Performance of the Dynamic Schedule for 8 Threads and for a number of different Chunk Sizes.

It is obvious that the performance that is achieved by the dynamic schedule is also very good. However, in this case the increment of the chunk size affects the speed-up differently than the static schedule. More specifically, the speed-up is about 6.5 when the chunk size is less than 800, but afterwards it is reduced significantly. When the chunk size is in the range of 800 to 2000 the speed-up tends to be equal to 6. However, for chunk size over 2000 the speed-up fluctuates between 5.5 and 5.9.

The GUIDED Schedule

The GUIDED schedule, which is the last schedule that has been used during this study, divides the iteration space in a number of pieces of work with decreasing size. This schedule is always used in conjunction with a chunk parameter. This chunk parameter determines the length of the smallest piece of work that is created after the division of the iteration space. Afterwards, this schedule uses the same approach with the dynamic, in order to assign each piece of work to a specific thread. As it was previously mentioned, this distribution method serves consecutively each piece of work to the first thread which is available. Below, in the Figure 5.7, the distribution method that is performed by the guided schedule is presented. [25]
The iteration space is divided in a number of pieces of work with decreasing size. The smallest piece of work has size equal to the chunk. Afterwards, the pieces are assigned to the first available thread.

In order for the performance of the Guided schedule to be checked, a similar approach with the previously mentioned tests have been used. More specifically, this schedule has been tested on 8 threads and for a number of different chunk size options. Below, in the Figure 5.8, the guided schedule’s performance has been presented.
The speed-up that is achieved by the guided schedule, fluctuates between the 6.5 and the 5. More specifically, when the chunk size is less than 100, the speed-up increases and reaches the 6.5 for chunk size equals to 100. Afterwards, by the increment of the chunk size, the speed-up follows a downward trend and tends to be equal to 5 for chunk size equals to 6000.

5.3.2 Performance Comparison among the Schedules

It is obvious that all the schedules which have been tested, achieve very good performance. However, a performance comparison among these schedules has also been performed in order for the one which achieves the best possible speed-up to be identified. This procedure takes also into account the different chunk size options of each schedule. In order for this procedure to be implemented, a graph which includes each schedule’s performance has been created. This graph is included in the Figure 5.9, which has been presented below.

![OpenMP Schedule's Performance](image)

Figure 5.9: The Performance Comparison among the Schedules.
The graph of the Figure 5.9 suggests that the performance which has been achieved by the previously mentioned schedules is almost identical when the chunk size is lower than 2000. However, the small differences among these curves can be explained by a few findings which describe the nature of the DO loop which has been executed in parallel. Initially, the performance of the static schedule is slightly worse than the performance of the other schedules. This finding can be explained by the fact that the static schedule does not assign the same amount of work to each thread. This has as a result few threads to finish their execution quicker than the others. Unfortunately, the whole application’s performance is influenced by the thread which has been assigned by the most amount of work [28]. This problem has not been encountered to the other schedules due to the fact that they support a better policy in order to distribute the iterations over the available threads.

In addition, the guided schedule achieves better performance than the dynamic. This can be explained by the fact that the guided, unlike the dynamic schedule, divide the iteration space in a number of pieces which include different amount of work. This has as a result when these pieces are distributed over the available threads, the best possible work distribution to be achieved. Under these situations, the best possible performance has been achieved by the guided schedule when it is used for chunk size equals to 100. This option has also been used in the FRAME model’s OpenMP version.

### 5.3.3 The Performance Evaluation of the OpenMP Version

As it was previously mentioned, by the completion of the performance comparison among the schedules, the guided with chunk size equals to 100 has been chosen. By this option, the FRAME model’s OpenMP version has achieved the best possible performance and this application’s final version has been reached. In addition, in order for the OpenMP version’s performance to be evaluated, this application has also been tested for a number of different threads. Due to the fact that the OpenMP version’s performance has to be checked in a shared memory architecture, the thread options that have been used are up to 8. In this section, the results of this procedure are presented and afterwards the reasons which explain this application’s performance are discussed.

**OpenMP Version’s Performance Analysis**

By the completion of the OpenMP version’s performance evaluation procedure, a graph which represents this application’s scalability, has been created. In this section, there is a presentation of this graph in the Figure 5.10, and afterwards the OpenMP version’s performance is described.
According to this graph, the FRAME model’s OpenMP version, achieves very good scalability, and reaches very good performance for the different thread options. This can be explained by the fact that the increment of the number of the threads, affects the speed-up, which also increases, and the performance, which remains close enough to the linear speed-up. More specifically, when this application is executed on two threads, the speed-up is almost identical to the linear speed-up (1.86). In addition, for the thread options which are lower than 5, this application’s performance is slightly lower than the linear performance. Finally, when the thread options are bigger than 5, the speed-up is reduced imperceptibly compared to the linear speed-up, and still remains in high levels.

The Reasons of the OpenMP Version’s Performance

According to this performance evaluation procedure, the OpenMP version performs very well. This achievement can be explained by a few factors that have been taken into account during the OpenMP version’s implementation procedure. Initially, the parallelisation strategy that has been performed, focuses only on the sections which spend large time for their execution. In addition, an experimental procedure has been implemented in order for the best possible work distribution to be achieved. During this procedure, the OpenMP version has been tested for a number of different schedules and the one which achieves the best possible performance has been chosen. By the completion of this procedure, the load balance among the threads has been achieved. Finally, the communications among the threads have been used carefully, in order for the sections that are executed sequentially inside the parallel section to be the least possible.
5.4 The Parallel Versions’ Performance Comparison

By the completion of the OpenMP and the HPF versions’ performance evaluation, a comparison between these two FRAME model’s parallel versions has also been performed. This procedure has been implemented, in order for the performance of these two parallel versions to be compared and afterwards, the reasons which explain this performance difference to be identified. In order for this aim to be achieved, a graph which includes the performance of these two parallel versions has been created. In addition, the performance difference of the OpenMP and the HPF versions has been analysed. Finally, the parallelisation strategies of these two parallel programming models have been compared in order for the performance difference between these two parallel versions to be explained. In this section, there is a detailed analysis of the previously mentioned procedure and afterwards, the findings of this procedure are discussed.

5.4.1 HPF and OpenMP Versions’ Performance Difference

As it was previously mentioned, in order for the performance difference between the two parallel versions to be evaluated, a graph has been created. This graph represents the OpenMP and the HPF versions’ performance, and also includes the linear speed-up. It is obvious that in this graph not only the performance difference between the two parallel versions is depicted, but also their performance difference from the linear performance is represented. Below, in the Figure 5.11, the previously mentioned graph is presented.

![Figure 5.11: The Performance Comparison between OpenMP and HPF.](image)

43
The graph of the Figure 5.11, suggests that the OpenMP version achieves better performance than the HPF version. It is obvious that when the number of the processors is increased, the speed-up of these two parallel versions also increases. However, the increment rate that is achieved by OpenMP is bigger than the HPF version’s. The OpenMP version’s curve has a slope of $6.5/8 = 0.8$, while for HPF the straight region is between 2 and 8 and has a slope of $(3.5 - 1.05)/6$, which is about 0.4. This different increment rate has also impact to the speed-up that is achieved by these two parallel versions. More specifically, when these two parallel versions are executed on two processors, OpenMP achieves speed-up almost equal to the linear speed-up while HPF performs as the serial. In addition, when the number of the processors is equal to 4, the speed-up that is achieved by HPF is equal to 2. However, OpenMP performs better and achieves speed-up equals to 3.5. Finally, the biggest performance difference occurs when the OpenMP and the HPF versions are executed on 8 processors. The speed-up difference is about 3, while OpenMP performs twice as much as HPF.

5.4.2 The Reasons for the Performance Difference

By the completion of the performance evaluation procedure which has been performed for each one of the parallel versions, a number of factors which affect the performance of the OpenMP and the HPF versions have been encountered. It is obvious that these factors are also responsible for the performance difference between these two parallel versions. More specifically, the parallelisation strategies that have been used by OpenMP and HPF respectively, affect the parallel versions’ performance differently. Initially, the HPF use a parallelisation strategy that distributes a large number of sections by the HPF_LOCAL directive, and achieves good enough execution time but it does not perform well. However, the OpenMP version’s parallelisation strategy distributes much less sections than the HPF and achieves the best possible performance. This difference can be explained by the fact that the OpenMP parallelisation strategy focuses only on the sections that have to be executed in parallel, while the HPF parallels a few more sections that do not spend large time for their execution. In addition, during the OpenMP version’s implementation procedure, the distribution strategy and the communications among the threads have been explicitly declared, while in the HPF version, the compiler is responsible for them.
5.5 OpenMP vs HPF model

As it was previously mentioned, in order for this project’s aims to be achieved, two parallel programming models have been studied and have been used. Initially, the Data Parallel model has been studied in order for the parallelisation methods that have been performed in the FRAME model’s HPF version to be identified. On the other hand, the Shared Memory parallel programming model has been used, in order for the FRAME model to be converted into an OpenMP parallel version. By the completion of this research study, a finding which suggests that these parallel programming models use different approaches in order to parallelise an application, has been reached. In this section, the methods that are used by the HPF and the OpenMP parallel programming models are presented and afterwards, they are compared.

Initially, the HPF and the OpenMP programming models, use different execution models. As it was previously mentioned, the HPF uses the single-threaded "HPF Global" model, unlike to OpenMP which uses a fork-join model [20] [24]. A comparison between these two execution models, suggests that the execution control flow that is performed by the single-threaded "HPF Global" is hard to be identified [30]. On the other hand, during the OpenMP execution, the threads that execute the application in parallel are synchronised at regular intervals, and the execution control flow can easily be identified [31]. In this study, the execution control flow has been used to describe the execution sequence with which a number of blocks of code are executed [31].

In addition, both HPF and OpenMP are consisted of a few compiler directives and are supported by the Fortran programming language [17] [1]. However, OpenMP is also supported by a few other programming languages such as the C and the C++ [1]. In order for the Shared Memory and the Data Parallel programming models to achieve an application’s parallel execution, they parallelise loops and independent code sections [30] [20]. The HPF model achieves the parallelisation of the independent sections by the usage of the extrinsic HPF_LOCAL model, which has also been used in this study [20]. However, in order for these parallelisations to be performed, they use different distribution strategies. The OpenMP supports the work distribution while the HPF performs the data distribution [24] [18]. Finally, both the HPF and the OpenMP applications are easy to be implemented [30].

As far as the performance concerned, OpenMP reaches better performance than HPF [30]. This can be explained by the fact that the performance of an HPF application depends on the compiler’s technology which is responsible for the data distribution [19]. However, the Shared Memory application achieves good performance only in shared memory architectures while the data parallel performs well not only in shared memory but also in distributed memory architectures [30]. Finally, an OpenMP application is more portable than an HPF application [30].
Chapter 6

Conclusions

During this study, an air pollution model, which is called FRAME, was used in order to be converted into a Shared Memory parallel application. This application had been implemented using Fortran and had already been parallelised using a Data Parallel model. The main aim of this study was to develop a new parallel version of the FRAME model which would be more portable than the available. Due to this fact, a new version of the FRAME model was implemented using the OpenMP parallel programming model. In order for the previously mentioned aim to be achieved, a few other goals were created and were achieved.

Initially, a new application which includes three different versions of the FRAME model was implemented. These three different versions are the serial, the HPF and the OpenMP. The main reason of this implementation was to support the HPF version’s good execution time and the portability of the OpenMP version in the same application. In order for this aim to be achieved, a few conditional compilation methods were used. These methods are described by the -D compiler flag, which informs the compiler about the source code sections that has to be taken into account, and the different directives that are used by HPF and OpenMP.

In addition, due to the fact that the HPF version achieves good enough execution time, an experimental procedure has been performed, in order for the parallelisation methods that are used by HPF to be identified. By the completion of this procedure, the execution that is achieved by the declaration of a few subroutines as HPF_LOCAL was identified. Afterwards, the contribution of a few arrays’ distribution in the previous execution was investigated.

During the implementation procedure, a profiling tool was used in order for the section that was executed in parallel to be identified. Afterwards, the OpenMP directives were used, and the previously mentioned section, which is a DO loop, was converted in a parallel DO loop. In order for this implementation to be achieved, a parallel section was declared out of the previously mentioned DO loop and an OpenMP schedule, which is responsible for the distribution of the iteration space over the available threads, was used. The completion of the FRAME model’s OpenMP version, was achieved by the
solution of a few problems that were encountered during the implementation procedure. Finally, a testing procedure also performed, in order for the correctness of this new application to be checked.

Finally, by the completion of the OpenMP version’s implementation, a performance evaluation procedure was performed, in order for the performance of the FRAME model’s two parallel versions to be measured. In addition, a performance comparison among the OpenMP schedules was implemented in order to be identified which one achieves the best possible performance. Afterwards, the performance of the OpenMP and the HPF versions was compared in order to be measured the performance difference between these two parallel versions. By the completion of this performance evaluation procedure, a few findings have been reached. Although the HPF version achieves good enough execution time, it does not perform well. However, OpenMP by the usage of the guided schedule achieves the best possible performance and ranges close enough to the linear speed-up.

In conclusion, the OpenMP version that has been implemented during this study, not only gives a portable feature to the FRAME model, but also performs better than the HPF version. However, the HPF version still achieves better execution time than the OpenMP version, due to the fact that can be executed on more nodes than OpenMP. Nevertheless, the increasing trend to be created Shared Memory architectures with more processors, make this application usable for the next years [28].
Chapter 7

Future Implementation

By the completion of this study, an OpenMP version of the FRAME model, which performs very well, has been developed. However, due to the fact that the High Performance system of the CEH is consisted of 27 Beowulf clusters, it would be interested to be checked the performance of a FRAME model’s parallel version which can take advantage of this distributed architecture. In order for this project’s scope to be achieved, the MPI parallel programming model can be used.

As it was previously mentioned, the HPF by the usage of the HPF_LOCAL directive achieves a data distribution which is similar to the distribution that is performed by the MPI parallel programming model. This approach is very convenient for the development of the FRAME model’s MPI version and sets this implementation straightforward. The data that have to be executed in parallel can be scattered over the available processors, and afterwards each one can work on his own data [21]. However, the sections that have to be executed in parallel should have been chosen more carefully, in order for the best possible performance to be achieved.
Appendix A

The Execution Times of the Performance Tests
A.1 The Execution Time Results of the Serial Version’s Experiments

<table>
<thead>
<tr>
<th>Section</th>
<th>Seconds</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>4.451</td>
<td>0.074</td>
</tr>
<tr>
<td>Kernel</td>
<td>10,764</td>
<td>179.4</td>
</tr>
<tr>
<td>Exit</td>
<td>4.769</td>
<td>0.079</td>
</tr>
</tbody>
</table>

Table A.1: The Execution Time Results of the FRAME Model’s Serial Version

A.2 The Execution Time Results of the HPF Version’s Performance Tests

<table>
<thead>
<tr>
<th>Processors</th>
<th>Seconds</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10,561.853</td>
<td>176.030</td>
</tr>
<tr>
<td>2</td>
<td>10,203.356</td>
<td>170.005</td>
</tr>
<tr>
<td>4</td>
<td>5,243.6087</td>
<td>87.3934</td>
</tr>
<tr>
<td>6</td>
<td>3,974.1684</td>
<td>66.2361</td>
</tr>
<tr>
<td>8</td>
<td>3,050.6831</td>
<td>50.8447</td>
</tr>
</tbody>
</table>

Table A.2: The Execution Time Results of the HPF Version’s Performance Tests
A.3 The Execution Times of the Performance Tests among the Schedules

<table>
<thead>
<tr>
<th>Chunk Sizes</th>
<th>Static Seconds</th>
<th>Minutes</th>
<th>Static Seconds</th>
<th>Minutes</th>
<th>Dynamic Seconds</th>
<th>Minutes</th>
<th>Guided Seconds</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1,638.687</td>
<td>27.3145</td>
<td>1,638.396</td>
<td>27.3060</td>
<td>1,632.221</td>
<td>27.2036</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1,691.121</td>
<td>28.1853</td>
<td>1,637.642</td>
<td>27.2940</td>
<td>1,626.501</td>
<td>27.1083</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>1,675.098</td>
<td>27.9183</td>
<td>1,649.975</td>
<td>27.4995</td>
<td>1,648.463</td>
<td>27.4743</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1,675.415</td>
<td>27.9237</td>
<td>1,672.092</td>
<td>27.8682</td>
<td>1,679.498</td>
<td>27.9916</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>1,715.901</td>
<td>28.5983</td>
<td>1,732.872</td>
<td>28.8812</td>
<td>1,720.506</td>
<td>28.6751</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>1,773.813</td>
<td>29.5635</td>
<td>1,878.859</td>
<td>31.3143</td>
<td>1,846.926</td>
<td>30.7821</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>1,803.577</td>
<td>30.0596</td>
<td>1,860.588</td>
<td>31.0098</td>
<td>1,914.892</td>
<td>31.9148</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>1,853.920</td>
<td>30.8986</td>
<td>1,818.154</td>
<td>30.3025</td>
<td>1,950.733</td>
<td>32.5122</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6000</td>
<td>2,309.839</td>
<td>38.4973</td>
<td>2,064.366</td>
<td>34.4061</td>
<td>2,030.955</td>
<td>33.8492</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.3: The Execution Time Results of the different Schedules’ Performance Tests

A.4 The Execution Times of the OpenMP Version’s Performance Tests

<table>
<thead>
<tr>
<th>Processors</th>
<th>Seconds</th>
<th>Minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10,773.553</td>
<td>178.8925</td>
</tr>
<tr>
<td>2</td>
<td>5,677.384</td>
<td>94.6230</td>
</tr>
<tr>
<td>4</td>
<td>2,981.324</td>
<td>49.6887</td>
</tr>
<tr>
<td>6</td>
<td>2,104.763</td>
<td>35.0793</td>
</tr>
<tr>
<td>8</td>
<td>1,625.799</td>
<td>27.0966</td>
</tr>
</tbody>
</table>

Table A.4: The Execution Time Results of the OpenMP Version’s Performance Tests
Appendix B

The Output of the FRAME model
B.1 The Output which Visualises the Ammonia Products’ Affection of UK

Figure B.1: The Ammonia Products’ Concentration and Deposition. *This map represents the affection of the UK by the ammonia products which are deposited by wet deposition.*
B.2 The Output which Visualises the Nitrogen Products’ Affection of UK

Figure B.2: The Nitrogen Products’ Concentration and Deposition.
*The affection of the UK by the nitrogen products which are deposited by dry deposition.*
The affection of the UK by the nitrogen products which are deposited by wet deposition.
B.3 The Output which Visualises the Sulphur Dioxide Products’ Affection of UK

Figure B.4: The Sulphur Dioxide Products’ Concentration and Deposition. 
*The affection of the UK by the sulphur dioxide products which are deposited by dry deposition.*
Figure B.5: The Sulphur Dioxide Products’ Concentration and Deposition.

*The affection of the UK by the sulphur dioxide products which are deposited by wet deposition.*
Bibliography

[1] Barbara Chapman, Gabriele Jost and Ruud Van Der Pas "Using OpenMP Portable Shared Memory Parallel Programming" Published in 2008


[3] Simon Courtney, Paul Burnett, Bill Heron "Using our cluster system nemesis 26 x 8 processors" Published in 2010, CEH Computer Support

[4] Nicolas Fournier "Development of an Atmospheric Transport Model Simulating Concentrations and Deposition of Reduced Nitrogen over the British Isles" Published in 2002, The University of Edinburgh


[9] Leesa Brieger "HPF to OpenMP on the Origin2000: A Case Study" Published in 2000, Geophysics Group, CRS4


[21] David Henty "MPP: Message-Passing Programming" Published in 2010, EPCC, The University of Edinburgh

[22] A K Ewing et al. "Writing Data Parallel Programs with High Performance Fortran" Published in 2011, EPCC, The University of Edinburgh


[25] Miguel Hermans "Parallel Programming in Fortran 95 using OpenMP" Published in 2002, School of Aeronautical Engineering, Universidad Politecnica de Madrid


[27] Charaka Palansuriya "Practical Software Development for Computational Scientists" Published in 2010, EPCC, The University of Edinburgh


[31] Fuerlinger, K., Moore S. "Capturing and analyzing the execution control flow of OpenMP Applications" Published in 2009, International Journal of Parallel Programming, Available at http://parlab.eecs.berkeley.edu/publication/234