PARALLEL COMPUTING FOR
FURNACE SIMULATIONS
USING DOMAIN DECOMPOSITION
Parallel computing for furnace simulations using domain decomposition

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K. F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College voor Promoties aangewezen,
op vrijdag 1 oktober 1999 te 16.00 uur

door Robbert Leonardus VERWEIJ
wiskundig ingenieur
geboren te Gouda
Dit proefschrift is goedgekeurd door de promotor:
Prof. Dr. Dipl.-Ing. K. Hanjalić

Samenstelling promotiecommissie:

Rector Magnificus
Prof. Dr. Dipl.-Ing. K. Hanjalić
Prof. dr. ir. P. Wesseling
Prof. dr. A.E.P. Veldman
Prof. dr. ir. F.T.M. Nieuwstadt
Prof. ir. C.J. Hoogendoorn
Dr. D.R. Emerson
Dr. ir. T.W.J. Peeters

voorzitter
Technische Universiteit Delft, promotor
Technische Universiteit Delft
Rijksuniversiteit Groningen
Technische Universiteit Delft
Technische Universiteit Delft
Daresbury National Laboratory, Groot Brittannië
Koninklijke Hoogovens

ISBN: 90-9013043-8
Illustratie omslag: Marco van der Wielen.
Samenvatting

Parallel rekenen gebaseerd op domein decompositie voor oven simulaties.

Eén van de beperkingen van de huidige computer-simulaties is dat de gebruikte modellen dermate complex zijn en de benodigde rekenroosters dermate groot dat de simulaties vaak veel rekentijd en geheugen kosten. Zeker bij complexe processen, zoals de verbranding van aardgas in glassmeltovens, zijn geavanceerde turbulentie-, verbrandings- en stralings modellen nodig, gecombineerd met fijne roosters, om alle belangrijke grootheden realistisch te voorspellen. Deze voorwaarden stellen op hun beurt weer hoge eisen aan de computers, waarop deze simulaties worden gedaan. Parallel rekenen wordt momenteel gezien als een veelbelovende techniek om de gewenste nauwkeurigheid in een acceptabele tijd te bereiken.

Dit proefschrift beschrijft een methode om de bestaande codes voor het simuleren van verbranding van gas in glassmeltovens te paralleliseren. Allereerst is de bestaande verschoven rooster aanpak vervangen door een niet-verschoven rooster aanpak. Er blijkt weinig verschil in convergentie-gedrag en oplossing. Het gebruik van multigrid leidt in de meeste gevallen tot een convergentie-gedrag onafhankelijk van het aantal roosterpunten.

De domein decompositie methode is onderzocht en geïmplementeerd om de effecten van domein decompositie op de kwaliteit van het convergentie-gedrag en de oplossing te bestuderen. De oplossingen blijken onafhankelijk van het aantal blokken en de topologie van de decompositie, alhoewel het convergentiegedrag enigszins verslechtert bij gebruik van meer blokken of decompositie in 'strips'. Dit gedrag is verklaard. Er is een methode ontwikkeld die het convergentie gedrag verbetert. Locale roosterverfijning is bestudeerd in combinatie met domein decompositie. Voor een gegeven nauwkeurigheid zijn er met locale roosterverfijning minder roosterpunten nodig dan met globale roosterverfijning; voor een goed gebruik van locale roosterverfijning zijn helaas wel ongestructureerde roosters nodig.

De domein decompositie is gebruikt om de code te paralleliseren, en blijkt hiervoor zeer geschikt. Er zijn communicatie-routines geschreven in platform-onafhankelijke code om het algoritme zo machine-onafhankelijk mogelijk te maken. Hiervoor zijn 'message-passing' paradigma's gebruikt. De parallele code draait op een groot aantal parallele computers en voor grote roosters (2 miljoen punten) zijn versnellingen van een factor 60 gemeten. Op toegewijde parallele systemen, zoals de CRAY T3E, is de communicatietijd verwaarloosbaar klein ten opzichte van de rekentijd. Het is moeilijk om alle processoren even veel werk te geven, wat tot aanzienlijke vertraging leidt.

Tenslotte is de huidige methode (parallel rekenen met locaal verfijnde blokken) vergeleken met de resultaten van voorgangers. Het blijkt dat er met de huidige code geen restricties meer zijn met betrekking tot het aantal roosterpunten en rekentijd. De resultaten komen overeen met voorgaande berekeningen en met metingen. Door gebruik te maken van deze technieken kunnen simulaties met meer realistische modellen en fijnere roosters in acceptabele tijd worden gedaan.
Summary

Parallel computing for furnace simulations using domain decomposition.

The complexity of the models and the size of the required grids restrict the current computer-simulations, since they require much computing time and memory. Especially complex processes, such as encountered in combustion of gas in glass melting furnaces, require advanced models for turbulence, combustion and radiation, in conjunction with fine grids, to obtain realistic predictions for all important quantities. These requirements, in turn, impose severe demands on computer resources. Parallel computing is currently seen as the promising route by which to achieve desired accuracy in acceptable turn-around time.

This thesis describes a method to parallelise existing turbulent combustion codes, focused on the combustion of gas in glass melting furnaces. First the staggered grid approach was replaced by a colocated grid approach. There was little difference in convergence behaviour and solution. Multigrid leads, in most cases, to a convergence behaviour independent of the number of nodes.

The domain decomposition method is investigated and implemented to study the effects of domain decomposition on the quality of the convergence behaviour and the solution. The solutions are independent of the number of blocks and the decomposition, although the convergence behaviour deteriorates slightly when using more blocks or a 'stripwise' decomposition. This behaviour is explained. A method is developed to decrease the deterioration of the convergence behaviour. Local grid refinement has been studied in combination with domain decomposition. For a given accuracy less nodes are needed with local grid refinement with respect to global grid refinement; for a decent use of local grid refinement unstructured grids are necessary.

Domain decomposition has been used to parallelise the code and is very suited for this purpose. Communication subroutines have been written in platform-independent code to ensure portability. For this the 'message-passing' paradigm has been used. The parallel code runs on a variety of platforms and for fine grids (2 million nodes) we have obtained a speedup of around 60. On dedicated parallel systems, like the CRAY T3E, the communication time is negligible with respect to the computing time. Load balance is extremely difficult and leads to severe delays.

Finally the current method (parallel computing with locally refined grids) has been compared to previous results. The current code has relieved the restrictions with respect to number of nodes and computing time. The results are in agreement with previous simulations and with experiments. Using these techniques, simulations can be done with more realistic models and finer grids in acceptable time.
Contents

Samenvatting v

Summary vi

1 Introduction 1
  1.1 Background ................................................ 1
  1.2 Previous work .............................................. 3
  1.3 Objectives .................................................. 5
  1.4 Outline ...................................................... 6

2 Mathematical model 7
  2.1 Introduction ................................................. 7
  2.2 Conservation equations ..................................... 7
  2.3 Models ....................................................... 9
    2.3.1 Turbulence modeling ..................................... 9
    2.3.2 Combustion modeling .................................... 11
    2.3.3 Radiation modeling ....................................... 15
  2.4 Boundary conditions ...................................... 16

3 Numerical methods 19
  3.1 Introduction ............................................... 19
  3.2 The Finite Volume Method ................................ 19
    3.2.1 Discretisation of the equations ....................... 19
    3.2.2 Boundary conditions .................................. 26
  3.3 Coupling the equations ................................... 26
    3.3.1 The SIMPLE method .................................... 26
    3.3.2 Underrelaxation ......................................... 29
    3.3.3 Convergence criteria ................................... 29
  3.4 The iterative solution procedure ......................... 31
    3.4.1 Linear system solvers .................................. 31
    3.4.2 Multigrid ................................................. 36
3.5 Radiation solution method ........................................... 41
3.6 Description of computed test cases ............................... 44

4 Domain decomposition .................................................. 51
4.1 Introduction .................................................................. 51
4.2 Reasons for using domain decomposition ......................... 51
4.3 Basic domain decomposition algorithm ......................... 53
4.4 Implementation of the domain decomposition ................. 57
  4.4.1 Creating the blocks .............................................. 57
  4.4.2 Connecting the blocks ......................................... 59
  4.4.3 Computing the blocks ......................................... 62
4.5 Convergence behaviour of the multi-block algorithm ....... 63
  4.5.1 The reason for convergence loss ............................ 63
  4.5.2 Influence of the number of blocks and topology ......... 66
4.6 Improvement of the multi-block convergence behaviour .... 74
  4.6.1 Variable-iterations increase .................................. 75
4.7 Local grid refinement criteria ...................................... 80
  4.7.1 The need for locally refined grids ............................ 80
  4.7.2 Possible routes to higher accuracy .......................... 81
  4.7.3 Refinement criteria ............................................ 83
  4.7.4 Coupling the local grid refinement to the domain decomposition 85
4.8 Results of local grid refinement criteria ......................... 89
  4.8.1 The influence of local grid refinement ..................... 89
  4.8.2 The influence of criteria on the local grid refinement .... 90
4.9 Conclusions ............................................................. 94

5 Parallelisation ............................................................. 95
5.1 Introduction ............................................................. 95
5.2 Differences between parallel and sequential computing .... 97
  5.2.1 Hardware differences ......................................... 97
  5.2.2 Software differences .......................................... 100
5.3 Great expectations .................................................... 102
  5.3.1 Speedup and efficiency ...................................... 102
  5.3.2 Single PE performance ........................................ 104
5.4 Parallelisation of domain-decomposition based codes ....... 105
  5.4.1 Solving radiative heat transfer in parallel FVM simulations 107
5.5 Ensuring portability and flexibility ................................ 109
5.6 Timings ................................................................. 109
  5.6.1 Speedup .......................................................... 109
  5.6.2 Influence of Nsolve parameter .............................. 110
  5.6.3 Effect of message passing protocol on speedup ......... 114
  5.6.4 Input/Output .................................................... 114
5.7 Load imbalance ......................................................... 116
CONTENTS

5.7.1 Performance per code part ............................. 116
5.7.2 Timings per code part for all blocks .................. 117
5.7.3 Effect of block decomposition on load imbalance .......... 119
5.7.4 Other reasons for load imbalance ...................... 121
5.8 Conclusions ............................................. 121

6 Furnace simulations .......................................... 123
6.1 Introduction ............................................ 123
6.2 Details of the computations ............................... 123
6.3 Computations without local grid refinement ................. 124
6.4 Computations with local grid refinement ................... 131
6.5 Conclusions ............................................. 137

7 Conclusions .................................................. 139
7.1 Introduction ............................................ 139
7.2 Numerical methods ....................................... 139
7.3 Domain decomposition ..................................... 139
7.4 Parallelisation ......................................... 141
7.5 Furnace simulations ..................................... 141

A Communication subroutines for parallel computing ............. 143
A.1 Subroutine parstart ..................................... 143
A.2 Subroutine parstop ..................................... 144
A.3 Subroutine parsend ..................................... 145
A.4 Subroutine parrecv ..................................... 146
A.5 Subroutine barglobal ................................... 147

Notations ...................................................... 150

Bibliography ................................................... 152

Stellingen ...................................................... 160

Propositions ................................................... 161

Curriculum vitae ............................................... 162
1 Introduction

1.1 Background

We don't like waiting. This might be an artifact of the western society. Whenever we go to more exotic countries we are usually advised to 'be patient' when confronted with what is called 'the mentality of people in warmer countries'. Time is less expensive in warmer countries, it seems. But still, the western society is what we live and work in. Waiting is annoying and costly and hence waiting time should in all cases be minimised. This is why the supermarket has several counters and why constructors build houses with a team of people rather than on their own. The result (paying for the groceries, or moving into your new house) is obtained faster.

In Computational Fluid Dynamics (CFD) the same reasoning is valid. Here the result to be obtained is the simulation of a complex flow problem. The people that don't like waiting are the scientists and (in industry) the people that have to make decisions based on these simulations. The difference between people in a supermarket and people waiting for CFD-simulations is that in CFD-simulations the fast obtaining of the result of a simulation might be of crucial importance. While waiting in a supermarket is usually just annoying (unless at Christmas), a decision, based on CFD-simulations, can sometimes be of vital importance when obtained in time. Think for instance of the hurricane-warning predictions in the United States. The hurricane-warnings are predicted by running complex weather prediction programs. These programs used to take days to produce an answer, in which case the program had proven to be very capable of foreseeing the hurricane that had struck two days ago! Would the results be available in a few hours, people could be informed and appropriate actions could be taken.

Parallel processing for CFD

The idea of using several processing units to achieve the results faster is called parallel computing. The cashiers at the supermarket and the team of construction workers are the 'processing units' that are all working on the same 'problem'. Getting the costumers out of the store, building the house. But they all perform their own ac-
tions, mostly independent of others. Sometimes communication between the different 'processing units' is needed to check if an action has been finished or can be started.

In parallel computing for CFD, the idea is similar. Several processing elements (PEs) are used to solve one computational task concurrently. These tasks might be completely identical and performed independent of each other (as is the case with the cashiers in the supermarket). This is called embarrassingly parallel computing. More common is that the tasks are similar, but some communication is needed to achieve the final answer. One could think of a costumer in a supermarket who has the total price of his chips computed at counter 1 and his beer computed at counter 2. Finally the two cashiers will have to communicate to arrive at the total amount for the costumer, who will pay once to one of the two cashiers.

The main job in performing a parallel computation is setting up this communication between the PEs, minimising the communication and checking the correctness of the results. It is now widely recognised that only parallel computing offers the potential of solving very large scale scientific problems, such as those encountered in CFD, but also in electro-magnetics, quantum-chemistry and many other computationally demanding fields, in an 'acceptable time' at a 'reasonable' cost.

**Turbulent combustion simulations**

Numerical computations of complex processes, such as encountered in glass melting furnaces, require advanced models of turbulence, combustion and radiation in conjunction with sufficiently fine numerical grids to resolve important small scale interactions in the areas of flame front, high shear and near solid walls.

Figure 1.1 shows the dependencies between different phenomena taking place in the combustion chamber. All these phenomena have to be taken into account for an accurate prediction. This, in turn, imposes severe demands on computer resources. Such computations are difficult to perform on typical computer facilities available in industry. Parallel processing is regarded nowadays as the promising route by which to achieve desired accuracy with acceptable turn-around time.

Industrial glass melting furnaces (as sketched in figure 1.2) are typical examples of large scale energy-intensive equipment whose performance could substantially be improved by means of advanced modelling and CFD, if adequate computer resources or computational techniques become more available for duly testing and optimisation. The predictions, obtained with such a model, can be used to improve the efficiency of the furnace, and to predict and reduce the emissions of various toxic gases.

Another application which comes within the realm of computation is the coupled solving of different models. One could think of the coupled solving of the turbulent combustion in a glass-melting furnace together with the flow and heat exchange within the glass melt. Other applications are the flow around an airplane wing and the movements of the wing as a result of this, which, in turn, will have an effect on the flow pattern of the air. These complicated combined coupled simulations might become feasible using high-performance computers.
1.2 Previous work

This section gives an overview of the work done in the last years on high performance computing and on furnace simulations. More detailed references are given in the subsequent chapters.

Among the first engineers to study the modelling of furnaces were Hottel and
Sarofim (1967). They could only predict overall averaged values for properties like temperature, velocities and concentrations. Later, Lucas and Lockett (1972) introduced variations of the flow and flame properties in one dimension, which were roughly valid for long narrow furnaces. The next year, Lockwood and Gosman (1973) introduced turbulent fluctuations for these simulations. Since then, both twodimensional (Spalding et al., 1975) and full furnace simulations have become possible.

This study is based on the work done by Post (1988), Wieringa (1992), Koster (1993), Peeters (1995), Boerstoel (1997). Post (1988) initially developed the furnace code for simulations of turbulent combustion in high-temperature gas fired glass-melting furnaces. Koster (1993) included a post-processor for predicting thermal NO formation. He also performed a validation of the code using the measurements performed at the IFRF (International Flame and Research Foundation). These measurements are described in Nakamura et al. (1993). This furnace geometry is still used to perform code validation. Wieringa (1992) extended the radiative heat transfer model and implemented the Discrete Transfer Method, allowing for much more detailed radiative heat transfer description. Peeters (1995) discussed several chemistry models suitable for industrial CFD applications. He also investigated the use of several probability density functions to model the turbulence-chemistry interaction. Finally, Boerstoel (1997) extended this code with more detailed submodels for combustion, soot formation and radiation. He also mentioned the importance of using very fine grids to resolve the very small length scales near the burner regions. Domain decomposition combined with local grid refinement was found to be a way to obtain accurate solutions, thereby avoiding excessive CPU-times.

However, it was also found that for a genuine solution to the problem of excessive CPU-times and memory demands, parallel computing was required. Hence this thesis focuses on the extension of the domain decomposition technique to parallel computers and the efficient use of these computers to solve turbulent reacting flow problems. Parallelisation for turbulent combustion simulation has gained increased popularity in the last five years. Risio et al. (1997) have performed a data-parallel approach to parallelisation to perform parallel simulations with a turbulent combustion code. However, they conclude that efficient compilers have to be available to enable a high degree of portability, which is currently not the case. Coelho and Carvalho (1993) mention the implementation of a domain decomposition approach for simulating flows in utility boilers. This program was later parallelised using a message-passing tool (Coelho, 1998). The parallelisation of the radiation is a separate topic. As this is not a convection-diffusion equation, the parallelisations needs to be considered separately. Different authors have suggested ways to parallelise the radiation simulations for the same model we have used. We refer to Novo et al. (1999) and Bressloff (1998). Coelho and Carvalho (1998) give a good description of the method, which is similar to the method we have used.

Domain decomposition is currently by far the most popular method for parallelisation of CFD-codes. This is not restricted to furnace-simulations and to universities.
1.3. Objectives

Now the background of this thesis has been outlined, the objectives of this thesis are a logical follow up of the previous studies. The main objective of this study is two-fold:

1. A contribution to the development of numerical techniques which are suitable for application in numerical CFD codes on parallel computers.

2. The application of these numerical techniques on the simulation and modeling of turbulent reacting flows.

These objectives were reformulated more detailed at the start of this study, leading to the following three objectives that form the core of this thesis.

1. The analysis, development and implementation of domain decomposition techniques on existing turbulent reacting flow simulation codes. Optimisation of this domain decomposition technique.

2. The extension of this technique to applications on parallel computers and the analysis of the domain decomposition on the efficiency of the parallel computations.
3. Investigation of the performance of these techniques to reacting flow simulations.

1.4 Outline

Chapter 2 describes the conservation equations that govern the transport of mass, momentum, species and energy in a combustion chamber. Models for turbulent flow, chemistry and radiation are introduced. The boundary conditions are discussed.

Chapter 3 explains the transformation of the system of partial differential equations to a finite number of difference equations. The radiation solution method is discussed. The algorithm available at the start of this study relies on the staggered grid arrangement. This has been replaced by the colocated grid distribution, to facilitate the implementation of block decomposition and multigrid. The colocated approach is explained and compared to the staggered approach. A set of test cases is presented to test the validity of the code for several different cases. Several solvers are discussed and tested against these test cases. Finally the multigrid approach is treated.

In chapter 4 a general domain decomposition technique on block-structured grids is given. Both staggered and colocated approaches can be handled. The blocks can be coupled in an arbitrary manner. The effect of block decomposition and solvers on the convergence is addressed. The use of complex solvers on structured grids results in a strong coupling between the nodes. Since the domain decomposition breaks this coupling in parts of the domain, convergence loss will occur. We propose an easy-to-implement technique to improve the convergence of an implicit multi-block solver. The improvement is very stable and yields good results on all considered cases. The issue of local grid refinement is discussed. Several refinement criteria have been tested to account for adaptive grid refinement.

Chapter 5 reports on the parallelisation of the code. A concise overview of the hardware and software available to create and execute parallel programs and some considerations on the possible gains of parallel computing are given. Then the parallelisation of the code is discussed in detail. This approach is very general and has also been used successfully to parallelise other CFD codes. The communication routines have served as a template for general purposes and are given in an appendix. Radiation simulations will be done in parallel. The relative gain of using more processors for a simulation (‘parallel efficiency’) is discussed. It is shown that CPU and memory restrictions do not limit the size of the simulations anymore.

Chapter 6 combines all the features that were investigated in this thesis for the simulation of turbulent reacting flows in industrial furnaces. The results of the calculations are compared to the results of previous studies and to measurements. Chapter 7 finally sums up the most important conclusions.
2 Mathematical model

2.1 Introduction

This chapter describes the conservation equations that govern the transport of mass, momentum, species and energy in a combustion chamber. Models for turbulent flow, chemistry and radiation are introduced. Finally the boundary conditions, needed to close the system of equations, are discussed.

2.2 Conservation equations

The equations below are stated in their time dependent form. For a more detailed derivation of these equations the reader is referred to standard textbooks on fluid dynamics and combustion like Williams (1985) or Rhine and Tucker (1991). The Einstein index notation is adopted, which means that a repeated index in a term implies summation over the rank of that dimension of the tensor. In our case, the dimension of space equals three. All quantities depend on space and time unless stated otherwise. The symbols and units are all listed in the nomenclature on page 150. The flow is considered to be incompressible, which means that the density is not influenced by the pressure. This assumption is roughly correct for Mach numbers $M_\infty < 0.3$. Very close to the gas inlet of a glass-melting furnace the local Mach number comes very close to 0.3, but this is only a very small part of the domain, and hence the assumption of incompressibility throughout the domain is justified. The density does vary with the temperature, though. Hence the flow is described as an incompressible, variable density flow.

Conservation of mass The continuity equation expresses the principle that no mass is produced nor annihilated in a gaseous reacting flow. It reads:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_j}{\partial x_j} = 0$$  \hspace{1cm} (2.1)
Conservation of momentum  The momentum equations govern the conservation of momentum of the flow in every direction. They read:

\[
\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_i u_j}{\partial x_j} = -\frac{\partial p}{\partial x_i} - \frac{\partial \tau_{ij}}{\partial x_j} + \rho g_i
\]

(2.2)

Here \( \tau_{ij} \) is the viscous part of the stress tensor. For Newtonian fluids, like water, air and natural gas under the conditions considered here, the stress tensor can be linearly related to the rate-of-strain tensor:

\[
\tau_{ij} \equiv -\mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) + \frac{2}{3} \mu \frac{\partial u_k}{\partial x_k} \delta_{ij}
\]

(2.3)

The bulk viscosity has been neglected.

Conservation of energy  Energy is contained in the flow as internal energy \( e \) and kinetic energy \( \frac{1}{2} u_i^2 \), per unit mass.

\[
E_{\text{total}} \equiv \frac{1}{2} u_i^2 + e = \frac{1}{2} u_i^2 + h - \frac{p}{\rho}
\]

(2.4)

Here the enthalpy has been introduced, defined as \( h \equiv e + \frac{\rho \varepsilon}{\rho} \), since it is convenient in combustion flows to solve the enthalpy equation rather than the equation for energy or temperature. This yields the following form for the enthalpy equation:

\[
\frac{\partial}{\partial t} \left( \rho h - p \right) + \frac{\partial \rho u_j h}{\partial x_j} = \frac{\partial}{\partial x_j} \left( \frac{\lambda}{c_p} \frac{\partial h}{\partial x_j} \right) + S_{\text{rad}}
\]

(2.5)

Here the Dufour effects are neglected, the Lewis number for all species is assumed to be equal to 1, and all compressibility- and viscous dissipation effects have been neglected. Also the kinetic energy is supposed to contribute insignificantly to the total energy. This assumption is justified since the chemical energy is much larger than the kinetic energy in turbulent combustion flames.

Conservation of species  During the combustion of natural gas many species will interact together in the chemical reactions that transform the fuel (Dutch natural gas, mainly methane and ethane) and oxidiser (air) to the products (mainly water and carbon dioxide). The mass fractions \( Y_i, i = 1, \ldots, N \) of all the species can change significantly during the numerous reactions that take place. For an accurate description of the combustion of natural gas and air, at least 40 species and 200 reactions should be taken into account (Warnatz, 1984). Conservation of all these chemical species can be cast into an equation for conservation of mass-fraction for each species:

\[
\frac{\partial \rho Y_i}{\partial t} + \frac{\partial \rho u_j Y_i}{\partial x_j} = \frac{\partial}{\partial x_j} \left( \Gamma_i \frac{\partial Y_i}{\partial x_j} \right) + S_i
\]

(2.6)

We assume Fickian molecular diffusion with unity Lewis number, which implies that the diffusion coefficient \( \Gamma_i \) will be equal for all species.
2.3. Models

State equations The set of equations stated above is not closed; since the density varies with temperature, an additional equation is needed. For this, it is assumed that the gas behaves like a thermally ideal gas.

\[
\frac{p}{\rho} = R^0T \sum_{i=1}^{N} \frac{Y_i}{M_i} \quad (2.7)
\]

Finally, the temperature is related to the enthalpy by the caloric equation of state:

\[
h = \sum_{i=1}^{N} \left( \int_{0}^{T} c_{p,i}(\theta) d\theta \right) Y_i \quad (2.8)
\]

This concludes the description of all the conservation equations and equations of state.

2.3 Models

The set of equations given above is closed and hence this system of conservation equations can be solved. The problem is, however, that these equations are nonlinear and coupled. Only for a limited number of cases, with simple geometries and laminar, non-reacting flows, analytical solutions can be obtained. For most flows of engineering interests, like turbulent flows and reacting flows, Roache (1972) showed that unique solutions to the Navier-Stokes equation most of the times fail to exist, because of the nonlinearity of the equations. This makes it very doubtful if one will ever be able to solve these equations analytically at all in their most general form.

Luckily, this generality is not required in many cases. At high Reynolds numbers viscosity might be neglected in parts of the domain, whereas at very high temperatures, the ratio of the turbulent characteristic time-scale and the chemical characteristic time-scale becomes so large, that the combustion process can be assumed infinitely fast. In practise, before even starting to discretise the equations, some levels of approximations have to be considered (Hirsch, 1988). This involves replacing the troublesome terms in the equations by modeled alternatives.

2.3.1 Turbulence modeling

At the high velocities that occur in industrial furnaces, the Reynolds number becomes so high that the flow becomes turbulent. This basically means that a wide range of velocity- and time scales exists in the flow. An overview on turbulence can be found in Hinze (1975).

In many engineering applications, one is mainly interested in the time-averaged components of the flow, and not in the fluctuating components. Also in the present study we are mainly interested in the time averaged behaviour of statistically stationary flames. Hence an averaging procedure is applied to obtain the laws of motion for
the mean quantities. The averaging has to be defined is such a way as to make the
influence of turbulent fluctuations explicit in the equations, so they can be modeled.
By virtue of Reynolds averaging, each variable is split into a time averaged part and
a fluctuating part:

$$\phi(x, t) \equiv \overline{\phi}(x) + \phi'(x, t) \quad \overline{\phi}'(x, t) \equiv 0$$  \hspace{1cm} (2.9)

We have applied ensemble-averaging in this study. For turbulent flows with variable
density it is convenient to replace Reynolds-averaged quantities occurring in the aver-
ging process by Favre-averaged quantities (Favre, 1969); this is a density-weighted
average, which avoids explicit density correlations in the Reynolds-averaged transport
equations. The Favre decomposition of a stochastic quantity $\phi$ is defined as:

$$\phi(x, t) \equiv \tilde{\phi}(x) + \phi''(x, t), \quad \tilde{\phi}(x) \equiv \frac{\rho(x)\phi(x)}{\rho(x)}$$  \hspace{1cm} (2.10)

Upon Reynolds-averaging the conservation equations (2.1-2.5), using the Favre de-
composed quantities, the equations become:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho \tilde{u}_i}{\partial x_j} = 0$$  \hspace{1cm} (2.11)

$$\frac{\partial \rho \tilde{u}_i}{\partial t} + \frac{\partial \rho \tilde{u}_i \tilde{u}_j}{\partial x_j} = - \frac{\partial \rho \tilde{u}_i \tilde{u}_j''}{\partial x_j} - \frac{\partial \tau_{ij}}{\partial x_i} - \frac{\partial \rho}{\partial x_i} + \tilde{p} g_i$$  \hspace{1cm} (2.12)

$$\frac{\partial}{\partial t} \left( \rho \tilde{h} - \rho \right) + \frac{\partial \rho \tilde{u}_j \tilde{h}}{\partial x_j} = \frac{\partial}{\partial x_j} \left( \frac{\rho}{c_p} \frac{\partial \tilde{h}}{\partial x_j} - \rho \tilde{u}_j'' \tilde{h}'' \right) + \tilde{S}_{\text{rad}}$$  \hspace{1cm} (2.13)

The species equation will be dealt with in the next paragraph.

The new unclosed terms appearing in the averaged equations are the velocity-
fluctuation correlations (also known as the Reynolds stresses $\rho \tilde{u}_i \tilde{u}_j''$) and the velocity-
scalar fluctuation correlations (the turbulent scalar fluxes $\rho \tilde{u}_i'' \tilde{h}''$). These terms de-
ote the influence of turbulence on the mean flow. Several approaches exist to find
expressions for these unclosed terms. One can derive transport equations for the
Reynolds stresses. This technique is called Reynolds Stress Modeling. The draw-
back of Reynolds Stress Modeling is that there are many unknown correlations in the
second-moment closure, which have to be modeled, and that more equations have to
be solved, leading to larger computing times and less robust convergence. Here the
Reynolds stress terms are modeled using the standard high-Reynolds number $k - \varepsilon$
turbulence model as quoted by Launder and Spalding (1972). The Reynolds stress and
turbulent fluxes are modeled as follows:

$$-\rho \tilde{u}_i'' \tilde{u}_j'' = \mu_t \left( \frac{\partial \tilde{u}_i}{\partial x_j} + \frac{\partial \tilde{u}_j}{\partial x_i} \right) - \frac{2}{3} \delta_{ij} \left( \rho \tilde{k} + \mu_t \frac{\partial \tilde{u}_k}{\partial x_k} \right)$$  \hspace{1cm} (2.14)

$$-\rho \phi'' \tilde{u}_i'' = \frac{\mu_t}{\sigma_\phi} \frac{\partial \phi}{\partial x_i}$$  \hspace{1cm} (2.15)
The eddy-viscosity $\mu_t$ is consequently defined as:

$$\mu_t \equiv \bar{p} C_\mu \frac{\tilde{k}^2}{\tilde{\varepsilon}}$$

(2.16)

Two additional equations have to be solved in this turbulence model (which is why this type of models is usually referred to as two-equation type model). One for the turbulent kinetic energy $\tilde{k}$ of the fluctuations, which yields a velocity scale $\sqrt{\tilde{k}}$ and one for the dissipation of turbulent kinetic energy $\tilde{\varepsilon}$, which defines a length scale $\tilde{k}^{3/2}/\tilde{\varepsilon}$. The equations for $\tilde{k}$ and $\tilde{\varepsilon}$ read:

$$\frac{\partial \tilde{k}}{\partial t} + \partial_x \tilde{u}_j \tilde{k} = \frac{\partial}{\partial x_j} \left( (\mu + \mu_t) \frac{\partial \tilde{k}}{\partial x_j} \right) + P_k - \bar{p} \tilde{\varepsilon}$$

(2.17)

$$\frac{\partial \tilde{\varepsilon}}{\partial t} + \partial_x \tilde{u}_j \tilde{\varepsilon} = \frac{\partial}{\partial x_j} \left( (\mu + \mu_t) \frac{\partial \tilde{\varepsilon}}{\partial x_j} \right) + \frac{\tilde{\varepsilon}}{k} \left( C_{\varepsilon 1} P_k - C_{\varepsilon 2} \bar{p} \tilde{\varepsilon} \right)$$

(2.18)

with $P_k$ the production of kinetic energy:

$$P_k = -\bar{p} u_i' u_j' \frac{\partial \tilde{u}_i}{\partial x_j}$$

(2.19)

The constants appearing in the $k$-$\varepsilon$ equations are taken to have the standard values: $C_\mu = 0.09$, $C_k = 1.0$, $C_\varepsilon = 1.3$, $C_{\varepsilon 1} = 1.44$ and $C_{\varepsilon 2} = 1.92$. After insertion of these modeled terms in the equation, all unclosed terms have been eliminated. The turbulent Prandtl number for enthalpy, $\sigma_h$, has been set to 0.9.

### 2.3.2 Combustion modeling

A complete description of the combustion in the furnace renders some severe problems. First of all, for a detailed description of the combustion of natural gas with air, over 200 reactions with more than 40 species have to be taken into account (Warnatz, 1984). This would require solving a stiff system of coupled non-linear PDE's and consequently require significant computer resources. Not all parameters in this reaction scheme are known with sufficient accuracy. A simplifying model is the way to circumvent this problem.

Secondly, there is the influence of combustion on turbulence and vice versa. The averaging procedure of the last section is not applicable here, since the average of the fluctuations of thermochemical quantities will not be zero. In combustion reactions the reaction rates can be expressed in an Arrhenius form as

$$w_f = -\rho Y_{f,\text{fuel}} Y_{\text{oxidiser}} A \exp \frac{-E}{RT}$$

(2.20)

The exponential dependence on the temperature results in a great difference between the time-averaged reaction rate and the rate of reaction based on averaged quantities. Hence the Reynolds averaging technique cannot be applied for mean reaction rates. Another averaging procedure will be discussed.
Conserved scalar approach The conserved scalar approach is a much used approach to model turbulent non-premixed diffusion flames, i.e. flames that are initiated by feeding fuel and oxidiser separately into the combustion chamber. A good review can be found in Williams (1985). The idea behind the model is to construct a set of conserved scalars, e.g. element mass fractions. Using so called Shtab-Zel’dovich coupling functions, all species can be related to each other through a single conserved scalar, which is usually called the mixture fraction. It denotes the 'mixedness' of fuel and oxidiser. By definition the mixture fraction equals 1 in the fuel inlet stream and 0 in the oxidiser inlet stream. If the combustion is adiabatic then also the enthalpy can be expressed as a linear function of this mixture fraction, and need not to be taken into account as an independent variable. The next paragraph discusses the treatment of non-adiabaticity in the combustion modeling.

The conservation equation for the mixture fraction reads, after Reynolds averaging:

\[
\frac{\partial \tilde{f}}{\partial t} + \frac{\partial \tilde{u}_j \tilde{f}}{\partial x_j} = \frac{\partial}{\partial x_j} \left( \rho \tilde{f} + \frac{\mu_t}{\sigma_j} \frac{\partial \tilde{f}}{\partial x_j} \right)
\] (2.21)

There is by definition no chemical source term in this equation since \( \tilde{f} \) is a conserved scalar. As explained in the previous paragraph, the Favre variance of \( f \), denoted by \( \tilde{g} \) is also of great importance because of the nonlinear influence of fluctuations. It is defined as \( \tilde{g} = f^2 - \tilde{f}^2 \), and the modeled transport equation for it reads:

\[
\frac{\partial \tilde{g}}{\partial t} + \frac{\partial \tilde{u}_j \tilde{g}}{\partial x_j} = \left( \rho \tilde{g} + \frac{\mu_t}{\sigma_g} \frac{\partial \tilde{g}}{\partial x_j} \right) + C_{1g} \mu_t \left( \frac{\partial \tilde{f}}{\partial x_j} \right)^2 - C_{2g} \rho \tilde{f} \tilde{g} \] (2.22)

where \( C_{1g} \) and \( C_{2g} \) are model constants with values 2.8 and 2.0 respectively. The turbulent Schmidt numbers are \( \sigma_f = \sigma_g = 0.9 \).

At the high temperatures prevailing in furnaces, the chemical timescales are much smaller than the turbulent timescales. The ratio is expressed by the Damköhler number. The Damköhler number in a furnace can be assumed very large, hence the combustion can be considered to take place instantly when oxidiser- and fuel particles meet (a so-called infinitely fast reaction), and the rate of combustion is limited by the mixing only, and not by the chemical kinetics.

The early attempts to model chemistry in a diffusion flame assumed that the fuel and oxidiser are burnt in one irreversible reaction the moment they mix. This is the so-called mixed-is-burnt or flame-sheet model (Burke and Schumann, 1928):

\[
\text{fuel} + \nu' \text{oxidiser} \rightarrow \text{products}
\]

Here \( \nu' \) is the stoichiometric coefficient. This model has some shortcomings (e.g. not being able to predict intermediate and radical species) which can only slightly be overcome by expanding this model to two-step reaction mechanisms (Post, 1988).
2.3. Models

In this thesis the more advanced constrained chemical equilibrium model by Bilger and Stårner (1983) has been used. It assumes that the reactions reach equilibrium infinitely fast, rather than combust completely. This makes it possible to include radicals and intermediate species in the simulation. Different equilibrium assumptions can be made; full-, partial or constrained equilibrium. Here the last model, the constrained equilibrium, has been used. This model, as well as a more extensive review of chemistry models for turbulent flames, can be found in Peeters (1995).

Modeling the influence of turbulence on the combustion To account for the turbulent fluctuations of the mixture fraction a so-called Probability Density Function (PDF) is used. If the combustion is adiabatic, all thermochemical quantities are directly related to the mixture fraction. Knowing the PDF of the mixture fraction immediately yields the mean value of any dependent quantity \( \phi = \phi(f) \), namely by:

\[
\bar{\phi} = \int_0^1 \phi(\zeta) P_f(\zeta) d\zeta
\]  

(2.23)

Here \( P_f \) is the Favre PDF of the mixture fraction. The main challenge is to find an expression for this PDF. Pope (1985) derives a transport equation for the PDF and solves it, mainly by means of Monte Carlo techniques. These techniques are very time and memory demanding and have hence mostly been applied to 2D simulations, although recently (Wouters, 1998) also 3D simulations are coming within reach, mainly because of the use of parallel computers.

Within the framework of this research the shape of the PDF was assumed from the start of the simulations: The so-called assumed-shape PDF modeling. Peeters (1995) sums up some of the most used PDF shapes. Most shapes are determined by the values of the the first two moments of the PDF, namely its mean (as computed by equation (2.21)) and its variance (2.22). In this work, the assumed \( \beta \)-shaped PDF has been used in its normalised form, as suggested by Boerstoel (1997):

\[
P_f(\zeta) = B^{-1}(a, b) \zeta^{a-1}(1 - \zeta)^{b-1} = \frac{\Gamma(a + b)}{\Gamma(a) \Gamma(b)} \zeta^{a-1}(1 - \zeta)^{b-1}
\]  

(2.24)

where \( \Gamma(a) \) is the gamma function defined as:

\[
\Gamma(a) = \int_0^\infty e^{-t} t^{a-1} dt, a > 0.
\]  

(2.25)

The PDF parameters \( a \) and \( b \) uniquely depend on the first and second moment:

\[
a = \frac{\bar{\phi}^2(1 - \bar{\phi})}{\bar{\phi}} - \bar{\phi}
\]  

(2.26)

\[
b = a \left( \frac{1 - \bar{\phi}}{\bar{\phi}} \right) = \frac{\bar{\phi}(1 - \bar{\phi})^2}{\bar{\phi}} - (1 - \bar{\phi}).
\]  

(2.27)
The validity of an assumed shape PDF will depend severely on its capability to capture the various shapes encountered in different parts of the furnace.

**Modeling of non-adiabaticity** Thermal radiation in a furnace plays an important role in the heat transfer to the furnace walls. This, together with the convective and conductive heat transfer, make the combustion strongly non-adiabatic. The problem that consequently arises is that the enthalpy is not a conserved scalar anymore, and can hence not be expressed as a linear function of the mixture fraction. It should be taken into account as an independent scalar. All thermo-chemical quantities can then be expressed as functions of both the mixture fraction and the enthalpy, $\phi = \phi(f, h)$. This would involve the solution of an additional equation for the enthalpy fluctuations, and consequently in the choice of a joint-probability density function for the mixture fraction and the enthalpy, rather than the simple univariate PDF that was suggested in the previous paragraph. All these problems have been avoided by suggesting a relationship between $f$ and $h$. In this way the enthalpy can still be expressed as a function of mixture fraction, but depending on the local degree of non-adiabaticity. Several different relationships have been suggested by Peeters (1995), all of the form $h = h(f, \tilde{f}, \tilde{h})$. The different models have been depicted graphically in figure 2.1. Figure 2.1 also shows the adiabatic case. All models incorporate the fact that for

![Figure 2.1](image)

**Figure 2.1:** Comparison of enthalpy loss models.

small fluctuations of $f$ the instantaneous enthalpy correlates very strongly to the instantaneous mixture fraction. Physically this means that there is a linear relationship
between \( f \) and \( h \) within a small interval \((\bar{f} - \delta f, \bar{f} + \delta f)\). Mathematically this means that the tangent of the function should be equal to the tangent of the adiabatic relationship for small fluctuations, since \( h(f = \bar{f}) = \bar{h} \). The first suggestion by Salooja (1978) was to simply extend this linear relationship for all values of the instantaneous mixture fraction. This yielded physically incorrect values for values of \( f \) close to 0 and 1, as can easily be seen. Megahed (1979) patched this problem by suggesting a piecewise linear profile that gave realistic prediction over the entire interval. Peeters (1995) then suggested a parabolic fit, which resembles the Salooja model for small fluctuations of \( f \) and resembles the Megahed model for larger fluctuations. Using this relationship, the univariate PDF of the mixture fraction only can be used to obtain the PDF-averaged values for the thermo-dynamical quantities.

### 2.3.3 Radiation modeling

One of the most important aspects of a glass-melting furnace is an optimal heat transfer to the glass. In large-scale industrial furnaces, radiation is a major contributor to the total heat transfer. At the high temperatures attained in furnaces (1800 K to 2300 K are typical values) 50% to 90% of all heat transfer to the walls of the furnace is caused by radiation.

The radiative heat transfer enters the equations via the source term of the enthalpy equation (2.5). The general equation for radiative heat transfer in gaseous media can be found in Modest (1993). A more concise overview of radiative heat transfer and its application to gas-fired furnaces can be found in Wieringa (1992).

If we neglect the scattering by the gas, the general radiative heat transfer equation along a line-of-sight is written as:

\[
i_i'(s) = i_i(0)\tau_i(0, s) + \int_0^s i_{i,\lambda}'(s') \frac{\partial \tau_i(s', s)}{\partial s'} ds'
\]  

(2.28)

Here \( i_i'(s) \) is the spectral radiant intensity in the direction of \( s \) within an interval \( \delta \lambda \) around \( \lambda \), \( \lambda \) is the wavelength of the beam in that direction. \( i_{b,\lambda}'(s) \) is the spectral black-body intensity. This spectral black-body intensity is directly related to the spectral emissive power by Planck's formula:

\[
\pi i'_{b,\lambda} \equiv E_{b,\lambda} = \frac{c_1 \lambda^{-5}}{\exp(c_2/\lambda T) - 1}
\]  

(2.29)

Here \( c_1 = 3.742 \times 10^{-16} \text{ W} \cdot \text{m}^2 \) and \( c_2 = 1.439 \times 10^{-2} \text{ K} \cdot \text{m} \) are respectively the first and second radiation constant. \( s' \) is the dummy integration variable defining the current location on the integration path between the points 0 and \( s \). The function \( \tau_i(s', s) \) is the spectral transmissivity of the gas from \( s' \) to \( s \), and is denoted by:

\[
\tau_i(s', s) = \exp \left( -\int_{s'}^s \kappa_i(s'') ds'' \right)
\]  

(2.30)
with $\kappa_\lambda(s)$ the spectral absorption of the medium. Boerstoeel (1997) concludes that without loss of accuracy a uniform absorption-emission coefficient can be used to account for gaseous radiation, which is what we did.

When the temperature and the gas composition are known, together with proper boundary conditions for the incident fluxes in the furnace walls, the spectral intensity can be solved for.

One of the major problems in solving $i'_\lambda(s)$ is the calculation of the transmissivity $\tau_\lambda(s', s)$. The wavelength-dependence of $\kappa$ makes the integration of (2.30) very difficult. A simplification can be achieved by assuming that the gas only emits and absorbs energy in narrow bands of the spectrum, as suggested by Grosshandler (1980) and used by Wieringa (1992). If it is further assumed that the absorption does not depend on the wavelength at all ($\kappa_\lambda = \kappa$), the gas is called grey, and the integration of (2.30) can be carried out immediately, yielding $\tau(s', s) = \exp(-\kappa | s' - s |)$. This will be used later in the explanation of the numerical method for solving the spectral intensity.

After the spectral intensity has been computed, the radiative source term can be expressed by the divergence of the radiative flux vector (Boerstoeel, 1997):

$$S_{\text{rad}}(s) = -\int_0^\infty \int_0^{4\pi} \frac{\partial i'_\lambda(s)}{\partial s} \cos \theta d\Omega d\lambda,$$

Here $S_{\text{rad}}(s)$ is positive if the beam loses energy to the gas. $\theta$ is the polar angle between the normal of the surface where the beam arrives and the direction of the beam itself. In principle one has to integrate over all solid angles. In practice, this invokes some problems, which will be dealt with in the next chapter, when discussing the numerical methods to solve this equation.

This concludes the description of all the models and equations used in the simulation of high-temperature gas-fired furnaces. The FURNACE code is also capable of predicting $NO_x$ formation (Koster, 1993) and take into account the effect of soot (Boerstoeel, 1997). These models have not been used in the present study, and hence will not be discussed here. The reader is referred to above references.

2.4 Boundary conditions

Finally the boundary conditions for the industrial glass-melting furnace, as also used in the previous studies (see chapter 1) are discussed. The other testcases used in this study are mentioned in section 3.6. For these cases the boundary conditions are discussed separately.
2.4. BOUNDARY CONDITIONS

Solid walls

The pressure on the boundary is obtained by linear extrapolation. For the mixture fraction and its variance the normal derivative has been set to zero.

Near solid walls the viscous effects become important, since the velocities in the flow tend to the velocity of the wall, because of the no-slip condition at the wall. This means that in this region of the domain the high-Re turbulence model (which was developed for $Re > 5000$) is not applicable here. The steep gradients, occurring in this region, would require a very fine grid in the near-wall region to resolve the flow pattern and the values for the turbulent quantities in the boundary region. Furthermore the turbulence model should be extended to take into account the effects that occur at low Reynolds number in the vicinity of the wall.

In a glass melting furnace the important regions (the flame zones) are located at a reasonable distance from the wall. Hence we can suffice with a less exact prediction of the near-wall behaviour, since we assume this near-wall behaviour will not influence the solution in the interior of the furnace too much. Hence it was decided to apply so-called wall functions to bridge the transition and laminar region near the wall. These wall functions assume that there exists a universal logarithmic velocity profile derived from the assumption that the shear stress $\tau_w \equiv (\mu \frac{\partial U}{\partial y})_{wall}$ is constant in the boundary layer. This universal profile has the following form:

$$U^+ \equiv \frac{1}{\kappa} \ln(Ey^+)$$  \hspace{1cm} (2.32)

where $\kappa = 0.42$ and $E = 9.8$ are universal constants and $U^+$ and $y^+$ are dimensionless distance to the wall, scaled with the friction-velocity $U_r$:

$$U^+ \equiv \frac{U_P}{U_r} \quad y^+ \equiv \frac{U_r y_P}{\nu}$$  \hspace{1cm} (2.33)

This friction velocity is related to the shear-stress by:

$$U_r \equiv \sqrt{\tau_w/\rho}$$  \hspace{1cm} (2.34)

For the turbulent kinetic energy the normal gradient to the wall has been set to zero. The near-wall turbulent quantities are obtained by assuming local equilibrium between production and dissipation in the boundary layer:

$$U_r \equiv \sqrt{C_p^{1/2} k_p}$$  \hspace{1cm} (2.35)

$$\varepsilon_P = \frac{U_r^{3/2}}{\kappa y_P}$$  \hspace{1cm} (2.36)

Finally for enthalpy a wall function for the heat flux to the wall is given by Khalil (1982) and Carvalho (1983):

$$q_{wall} \equiv \rho U_r \frac{h_p - h_{wall}}{\sigma_h U^+ + P_j}$$  \hspace{1cm} (2.37)
\[ P_j \equiv \frac{\pi/4}{\sin(\pi/4)} \sqrt{\frac{26.0}{\kappa}} \left( \frac{\sigma_h}{\sigma_{h, lam}} \right)^{1/4} (\sigma_{h, lam} - \sigma_h) \]  

(2.38)

The subscript \( P \) refers to the value of a quantity in the near-wall point. This causes some restrictions to the location of the first grid node, since it should be located in the fully developed turbulent boundary layer. However, the boundary layer will be relatively small compared to the size of the furnace, so this constraint is not very restricting.

**Inlet boundary**

The furnace types under consideration here all have two inlets: One for the oxidiser (usually air) and one for the fuel (usually natural gas). The pressure is obtained by linear extrapolation. For the velocities a uniform profile was used. The mixture fraction was set to 1 in the fuel stream and 0 in the air stream. The mixture fraction fluctuations are set to zero in the inlets.

For the turbulence quantities, the following inlet conditions are given:

\[ k_{in} = \frac{3}{2} I_t^2 |u_{in}|^2 \]  

(2.39)

\[ \varepsilon_{in} = \frac{G_{\mu}^{3/4} k_{in}^{3/2}}{0.03 L_h} \]  

(2.40)

Here \( I_t \) is the turbulence intensity, taken to be 0.1 in the air inlet and 0.05 in the fuel inlet. \( L_h \) is the macro length scale of the turbulence, taken to be equal to the hydraulic inlet diameter. With this definition the turbulence intensity is taken to be some 22% (\( \sqrt{3/2} \)) higher than in the regular definition. These numbers are based on isothermal experiments and calculations done by Post (1988).

**Outlet boundary**

The outflow boundary poses some problems in the simulation, since nothing is known at all about the variables or their fluxes before the computation. However, if the outflow boundary is sufficiently far downstream, the derivatives of all quantities in the direction of the outflow will be zero. In this case zero gradients can be assumed for all variables. Since the absolute pressure level is needed in the equation of state, the pressure is assumed to be equal to the (fixed) atmospheric pressure in one point of the outlet plane.

**Symmetry boundary**

At the symmetry plane all variables are assumed to have a zero gradient normal to the symmetry plane. The velocity component normal to the symmetry plane has zero Dirichlet boundary conditions.
3 Numerical methods

3.1 Introduction

This chapter describes the transformation of the system of partial differential equations to a finite number of difference equations. The radiation solution method is discussed. The algorithm available at the start of this study relies on the staggered grid distribution. This has been replaced by the colocated grid distribution, to facilitate the implementation of block decomposition and multigrid. The colocated approach is explained and compared to the staggered approach.

A set of test cases is presented to test the validity of the code for several different cases. Several solvers are discussed and tested against these test cases. Finally the multigrid approach is treated.

3.2 The Finite Volume Method

There are a number of ways to transform the system of partial differential equations into a finite number of difference equations. In Computational Fluid Dynamics, the Finite Difference Method (FDM) the Finite Volume Method (FVM) and the Finite Element Method (FEM) are most commonly used. Other methods include spectral methods and cellular automata, but they are not used in the present study.

The main reason for using the Finite Volume method is that it starts from the integral form of the transport equations. Hence local conservation is always satisfied in every control volume. It can be applied to both structured and unstructured grids, having both the flexibility of FEM and the ease of programming of FDM. All terms have explicit physical meaning, making interpretation of the discretised system straightforward. A drawback of the method is that the implementation of higher order schemes can become quitecumbersome.

3.2.1 Discretisation of the equations

In the Finite Volume Method the discretisation takes place in three steps:
1. Definition of a finite number of control volumes (CVs) covering the geometry of interest.

2. Arrangement of the location of the variables on the grid

3. Integration of the equations over all control volumes

**Definition of control volumes** The computational domain is divided into a finite number of $N_x \times N_y \times N_z$ disjunct CVs by defining a grid onto the computational domain. In our case the domain is always chosen to be a rectangular box, and all grid lines are parallel to the boundaries of the computational domain. As a result all CVs are also rectangular, and the grid is called Cartesian and rectilinear.

**Arrangement of unknowns on the grid** After dividing the domain into CVs, the location of the unknowns on the grid, (i.e. the place on the domain where the values of the unknowns are to be computed) must be selected. There are two main options here, the *staggered* approach and the *colocated* approach. Figure 3.1 shows a two dimensional view of both approaches, the locations of the unknowns on the grid and the definition of the CVs. Note that in the colocated approach there is just one cell type (for radiation a separate grid is applied, as shown later).

![Figure 3.1: Left Staggered grid Right Colocated grid; Legend A: scalar variables cells (staggered) all variables (colocated), B: U-velocity cell (staggered), C: V-velocity cell (staggered) ](image)

The staggered grid approach was suggested by Harlow and Welch (1965) and has been much adopted for CFD computations. The main reason for this was the natural, strong coupling of the pressure- and velocity field, avoiding all kinds of velocity-interpolations that lead to a decoupling of the pressure field (the infamous *checkerboard* pressure field). Patankar (1980) elaborates this fact in more detail.
3.2. THE FINITE VOLUME METHOD

A large drawback of the method is the fact that a computer programmer now has to store four different grids (three for the velocity components and one for the scalar quantities). Especially for complex multi block geometries with local grid refinement and multigrid this greatly hampers the interpolation and programming.

The colocated approach, in which all independent quantities are stored at the same location (explaining the name of the method), proves a much more natural way of arranging the nodes on the grid. Rhie and Chow (1983) realised that the source of the checkerboard pressure field lies in the interpolation of the pressure for the momentum equations. They added a correction term to this interpolation term. A detailed review about the use of colocated grids in CFD is given by Ferziger and Perić (1996). A comparison between staggered and numerical grids can be found in Perić et al. (1988) or Shih and Tan (1989). Both conclude that both solution methods yield almost identical convergence rate, dependency on underrelaxation parameters, computational effort and accuracy.

At the start of this study only a staggered version of the code was available. Boerstoel (1997) gives a derivation of the equations for this approach. For the multi block and multi grid extensions to the code, the colocated grid arrangement was thought to be more convenient. Therefore it was decided to transform the entire code to the colocated grid arrangement. Also the treatment of the boundary conditions becomes much more natural in this approach. It should be noted here that to the author’s opinion one approach is not 'worse' of 'better' than the other for the current simulations. It is merely a matter of convenience.

To test the proper transformation of the code and to see the differences occurring because of the new boundary condition treatment, as explained in section 3.2.2, two furnace simulations, described in section 3.6 were performed on the same grid, but with both the staggered and the colocated approach. The grid consisted of $32 \times 50 \times 50$ CVs. A qualitative comparison is given in figure 3.2. Table 3.1 gives a quantitative comparison. There are some differences in the solutions, but these are caused

<table>
<thead>
<tr>
<th></th>
<th>Staggered</th>
<th>Colocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average temperature</td>
<td>1674K</td>
<td>1732K</td>
</tr>
<tr>
<td>Average bath load</td>
<td>60.6 kW/m²</td>
<td>64.6kW/m²</td>
</tr>
<tr>
<td>$O_2$ flux outlet</td>
<td>$2.6 \times 10^{-3}$ kg/s</td>
<td>$2.0 \times 10^{-3}$ kg/s</td>
</tr>
</tbody>
</table>

by the different implementation of the boundary conditions. Especially the inlet conditions for the turbulent quantities are very sensitive to the solution. This was also found in previous work by Post (1988). Furthermore the convergence behaviour and underrelaxation parameters were all similar. It can be concluded that there are only some expected differences in the solutions and that the overall convergence behaviour of both approaches is similar.
Integration of the equations over all control volumes All partial differential equations derived in the last chapter are convection-diffusion equations. In their generic form they can be written as:

\[
\frac{\partial \rho \varphi}{\partial t} + \frac{\partial J_{\varphi,j}}{\partial x_j} = \rho S_{\varphi}
\]  

(3.1)

where \( J_{\varphi,j} \) denotes the total flux of quantity \( \varphi \) in the \( x_j \) direction. The total flux consists of the sum of the convective and the diffusive flux:

\[
J_{\varphi,j} = J_{\varphi,j}^C + J_{\varphi,j}^D = \rho u_j \varphi - \Gamma_{\varphi} \frac{\partial \varphi}{\partial x_j}
\]  

(3.2)

This equation is integrated over each CV. In the colocated grid approach we only have to integrate the equation once for all three velocity components, whereas in the staggered grid approach the equations have to be integrated for every variable separately, since they all have differently positioned CVs. Using Gauss' theorem (\( \int_{CV} \frac{\partial u_j \varphi}{\partial x_j} dV = \int_{CV} \int_{\partial CV} u_j \varphi n_j dS \), with \( n_j \) the normal on surface \( S \) pointing outwards), we can rewrite the volume integral for the fluxes as a surface integral. The integral
over the CV-surfaces is then evaluated using the midpoint rule approximation, which is second-order accurate.

![Figure 3.3: Notation conventions for all control volumes](image)

All terms in equation (3.1) will now be discussed separately.

The diffusive flux is approximated using the central-differencing scheme (CDS); for the east face, for instance, the expression becomes:

$$\int_{S_e} J_{\varphi,j}^D dS \approx -\Gamma_{\varphi,e} \left( \frac{\varphi_E - \varphi_P}{x_E - x_P} \right) \Delta y \Delta z$$  \hspace{1cm} (3.3)

The expression for the convective term for the east face is as follows:

$$\int_{S_e} J_{\varphi,j}^C dS \approx \varphi_e \int_{S_e} \rho u_j n_j dS \approx \varphi_e \rho_e U_e \Delta y \Delta z = \varphi_e m_e \Delta y \Delta z$$  \hspace{1cm} (3.4)

where $U_e$ stands for the value of the velocity component in $x$-direction on the east face and $m_e$ stands for the mass flux through the east face (in kg/m²s). Note that if $\varphi$ equals any velocity component, this convective term will be non-linear. Above equation implicitly includes the linearisation of the convective terms in that case. Since the values of all quantities are stored at the cell centre an interpolation technique is needed to obtain the values on the cell faces. The harmonic mean is used for the diffusivity $\Gamma$, while for the density the arithmetic mean is used.

In the literature much attention has been given to the treatment of the interpolation of the quantity $\varphi_e$. Poor interpolation of this term leads to slow convergence or
poor resolution, as the implementation of a poor scheme will introduce severe numerical errors. We have applied several schemes for the convection interpolation, namely first order upwind, (UDS), central differencing scheme (CDS) and the hybrid difference scheme (Patankar, 1980). A second order upwind scheme was also implemented, since Leonard and Drummond (1995) argue that the hybrid scheme can lead to qualitatively incorrect solutions in case of recirculating flows, and the CDS is not always stable on coarse grids. The second-order schemes that were implemented were all Total Variation Diminishing (TVD) schemes. For a detailed description of TVD schemes we refer to Hirsch (1990). The idea behind the TVD schemes is that a non-linear higher-order interpolation scheme is used to determine the value on the cell-face. To avoid the occurrence of unrealistic interpolated values this value is clipped between the minimum and maximum values using so called limiters. A review on limiters is given by Zijlema (1996). We have implemented the minmod, Roe’s Superbee and Van Leer limiters (Zijlema, 1996).

The expression for \( \varphi_e \) for UDS, CDS and hybrid schemes can be found in Patankar (1980), the expression for the higher order upwind scheme is:

\[
\varphi_e^{\text{HO}} = \begin{cases} 
\varphi_P + \frac{1}{2} \Psi \left( \frac{\varphi_P - \varphi_W}{\varphi_E - \varphi_P} \right) (\varphi_E - \varphi_P) & m_e \geq 0 \\
\varphi_E + \frac{1}{2} \Psi \left( \frac{\varphi_E - \varphi_P}{\varphi_{EE} - \varphi_E} \right) (\varphi_{EE} - \varphi_E) & m_e < 0
\end{cases} \tag{3.5}
\]

with \( \Psi(r) \) being one of the following (non-linear) limiters:

\[
\Psi(r) = \begin{cases} 
\max(0, \min(r, 1)) & \text{minmod} \\
\max(0, \min(2r, 1), \min(r, 2)) & \text{Roe Superbee} \\
(r + |r|)/(1 + r) & \text{Van Leer}
\end{cases} \tag{3.6}
\]

All second-order schemes are implemented using the deferred correction approach (Khosla and Rubin, 1974), ensuring the diagonal dominance of the resulting system and retaining the band structure of the resulting influence matrix. In practise this means that only the first-order upwind part of the variable is used to build the system and the remaining part is added to the right hand size of the matrix equation. The convective terms that are transferred to the RHS are evaluated using the known ‘old’ values as approximations. In the converged case the lower-order approximation cancels out and the higher-order approximation remains.

\[
\varphi_e = \frac{\varphi_e^{\text{UDS}}}{\text{In matrix}} + \frac{(\varphi_e^{\text{HO}} - \varphi_e^{\text{UDS}})}{\text{In RHS}} \tag{3.7}
\]

The source term is integrated over the volume, rather than applying Gauss’ theorem. The cell-centre values of all relevant quantities are used to obtain integrated values.

\[
\int_{CV} \rho S_\varphi dV \approx \rho_P(S_C + S_P\varphi_P)\Delta x\Delta y\Delta z \equiv \rho_P(S_C + S_P\varphi_P)\Delta V \tag{3.8}
\]
3.2. The Finite Volume Method

The source term has been linearised for iteration purposes. Peeters (1995) shows how the implementation of the source terms is done.

Finally, the time derivative is handled using the first-order implicit Euler scheme with a time step $\Delta t$. This scheme is unconditionally stable. Quantities evaluated at the old time step $t - \Delta t$ are marked $\varphi^{t-\Delta t}$. All other quantities are evaluated at the current time step.

$$\int_{CV} \frac{\partial \rho \varphi}{\partial t} dV \approx (\rho_P \varphi_P - \rho_P^{t-\Delta t} \varphi_P^{t-\Delta t}) \frac{\Delta V}{\Delta t}$$

(3.9)

After collecting all terms, we obtain the following discretised form of differential equation (3.1) for grid point P:

$$a_P \varphi_P = \sum_{nb}(a_{nb}\varphi_{nb}) + b_P$$

(3.10)

where the summation runs over the neighbouring grid points E, W, N, S, T and B, as shown in figure 3.3. This is the well-known 7 point stencil. The expressions for $a_P$ and $b_P$ are:

$$a_P = \sum_{nb}(a_{nb}) + \left(\frac{\rho_P}{\Delta t} - \rho_P S_P\right)\Delta V$$

(3.11)

$$b_P = \left(\frac{\rho_P^{t-\Delta t} \varphi_P^{t-\Delta t}}{\Delta t} + \rho_P S_C\right)\Delta V$$

(3.12)

The expressions for $a_{nb}$ depend on the scheme used for the convective terms. We define the cell Péclet number on the east wall as:

$$Pe_e \equiv \frac{\rho_e u_e \Delta x_e}{\Gamma_e}$$

(3.13)

where $\Gamma_e$ is the conductivity for variable $\varphi$ on the east wall, $u_e$ the normal velocity through the east wall and $\Delta x_e$ the distance between the nodes P and E, and similar expressions for all other walls. The precise expressions for the neighbour coefficients for the UDS, CDS and hybrid scheme (see also Patankar (1980)) is given by ($D_e$ and $C_e$ are respectively the diffusive and convective flux through the east face):

$$a_w = D_w A(||Pe_w||) + \max(C_w, 0)$$

$$a_n = D_n A(||Pe_n||) + \max(-C_n, 0)$$

$$a_s = D_s A(||Pe_s||) + \max(C_s, 0)$$

$$a_t = D_t A(||Pe_t||) + \max(-C_t, 0)$$

$$a_b = D_b A(||Pe_b||) + \max(C_b, 0)$$

The function $A(||Pe||)$ reads:

$$A(||Pe||) = \begin{cases} 1 & \text{Upwind scheme} \\ 1 - 0.5|Pe| & \text{Central scheme} \\ \max(1 - 0.5|Pe|, 0) & \text{Hybrid scheme} \end{cases}$$

(3.14)
3.2.2 Boundary conditions

To close the system of difference equations, given by (3.10), boundary conditions are imposed by prescribing either the gradient or the values of all quantities on the external boundaries, as explained in section 2.4.

In the staggered-grid version of the code, the boundary conditions were implemented through the use of virtual cells, located outside the computational domain. In the collocated version all boundary nodes are located on the boundary of the computational domain. This leads to a more natural implementation of boundary conditions.

3.3 Coupling the equations

The previous section explained how to solve a difference equation when the density, viscosity, diffusivity, source term and velocity field are known. However, if at least one of these is not known, or is given by another equation, some coupling procedure has to be found to resolve the coupling between the different equations.

The coupling of the pressure and the velocity field is the most cumbersome, since there is no explicit equation to solve for pressure. Patankar and Spalding (1972) introduced an iterative method, called SIMPLE (Semi Implicit Method for Pressure Linked Equations), to derive an equation for the pressure and a method to solve the coupling between the pressure and the velocity field. This coupling is treated in section 3.3.1.

An iterative procedure has also been used to treat the coupling between the remaining equations in the system. In practice the equation derived for variable ϕ (with ϕ being any independent variable for which a convection-diffusion equation is solved) is treated as if it only contained ϕ as unknown term, using the latest available values for all other quantities as best guesses. All equations can then be solved in turn. After all variables have been updated this procedure can be repeated with the new values as (improved) guesses. This process can be repeated until the change in the values of all relevant quantities is below some required tolerance. The determination of convergence criteria is treated in section 3.3.3. This approach leads to three iteration loops which are depicted in figure 3.4.

1. The cycle just mentioned, looping over all equations, and building the systems using the latest available values for all other variables. We call these cycles the outer iterations.

2. The iteration procedure that treats the pressure-velocity coupling, the SIMPLE-algorithm.

3. For every equation a linearised system has to be solved. This, as will be discussed in section 3.4.1, is done using an iterative solver. The iterations performed to solve the system for a specific variable ϕ are called inner iterations.
3.3. Coupling the equations

![Flow chart for the Furnace code with iteration loops](image)

**Figure 3.4**: Flow chart for the Furnace code with iteration loops

Alternative coupling methods can also be applied. One could, for instance, build the systems with all velocity components as unknowns and solve for all the velocity components simultaneously. This will lead to a better coupling between the equations. However, the matrices arising in these computations will be much larger and is hence often prohibited by the size of the memory.

3.3.1 The SIMPLE method

Ferziger and Perić (1996) give the derivation of the equations for colocated grids. We will repeat here the highlights of the SIMPLE-method for colocated grids.

First the linearised momentum equations are solved in turn. The new mass fluxes
through CV faces are calculated (viz. figure 3.3 for the notations), e.g. for the east face:

$$ m_e^* = \rho_e U_e^* \Delta y \Delta z $$

(3.15)

where the * denotes the intermediate velocity field, satisfying the momentum equations but not the continuity equation (unless the solution is completely converged). The cell-face velocity component $U_e^*$ is computed using a procedure first suggested by Rhie and Chow (1983):

$$ U_e^* = \overline{(U^*)_e} + \Delta V \ast \left( \frac{1}{a_P} \right)_e \left[ \left( \frac{\partial p}{\partial x} \right)_e - \left( \frac{\partial p}{\partial x} \right)_e' \right] $$

(3.16)

The overbar denotes linear interpolation and the tilde the arithmetic average. $a_P$ denotes the main diagonal component of the equation for the velocity (in the colocated approach this value is equal for all three velocity components). The cell-face derivative is calculated using central differences. The square brackets denote a correction term, which is zero when the pressure varies linearly or quadratically; it is designed to smooth out artificial oscillations which may develop into the decoupling of the pressure field ("checkerboard pressure field").

The so calculated mass fluxes do not satisfy the continuity equation, so they need to be corrected by adding a velocity correction $U_e'$ to the intermediate velocity field. Substituting this expression for the total velocity $U_e^* + U_e'$ in (2.2) (the momentum equation) and cancelling the parts that satisfy the momentum equation, we can obtain an expression for the relation between the pressure correction and the velocity correction. (Verweij (1994) shows that it is also possible to obtain other expressions for the pressure-velocity coupling, but this is the standard approach):

$$ U_e' \approx -\Delta V \left( \frac{1}{a_P} \right)_e \left( \frac{\partial p'}{\partial x} \right)_e $$

(3.17)

An equation for the pressure correction can be obtained by demanding that the corrected mass fluxes should satisfy the continuity equation (2.1). This leads to a Poisson-like equation for the pressure correction, where the right-hand side consists of the mass deficit (see Ferziger and Perić (1996) for details).

Finally this pressure-correction field is used to obtain the velocity corrections. In the basic approach the pressure correction is also used to correct the pressure field, although it is argued that the pressure correction field might not be suited to do so. Patankar (1980) has come up with an improved version of SIMPLE where a separate equation for pressure is derived similar to the way the pressure correction equation is derived.

The complete SIMPLE algorithm involves the sequential solving of the three momentum equations, followed by the solution of the pressure correction equation. The mass fluxes are also corrected and used to obtain the system for the next iteration, as
the corrected velocities and pressure will probably not satisfy the moment-equations anymore. This procedure is repeated until the pressure correction is smaller than a user defined tolerance.

### 3.3.2 Underrelaxation

The non-linear nature of the equations requires the use of underrelaxation of the equations to keep the iteration procedure stable. The difference between the value \( \varphi_{old} \) from the previous outer iteration and the newly computed value after solving the equation (\( \varphi_{new} \)) might be very large. The idea is to avoid these large differences between consecutive outer iterations by only adding a percentage \( \alpha \) of this difference to the old solution, rather than completely replacing it. The difference between the old and newly computed value is, starting from (3.10):

\[
a_P \varphi_P^{new} = \sum_{nb} (a_{nb} \varphi_{nb}^{new}) + b_P \Leftrightarrow \varphi_P^{new} = \frac{\sum_{nb} (a_{nb} \varphi_{nb}^{new}) + b_P}{a_P} \Leftrightarrow \\
\varphi_P^{new} - \varphi_P^{old} = \frac{\sum_{nb} (a_{nb} \varphi_{nb}^{new}) + b_P}{a_P} - \varphi_P^{old}
\]

The underrelaxed system of equations then becomes:

\[
\varphi_P^{new} = \varphi_P^{old} + \alpha \left( \frac{\sum_{nb} (a_{nb} \varphi_{nb}^{new}) + b_P}{a_P} - \varphi_P^{old} \right) \Leftrightarrow \\
\frac{a_P}{\alpha} \varphi_P^{new} = \sum_{nb} (a_{nb} \varphi_{nb}^{new}) + b_P + (1 - \alpha) \frac{a_P}{\alpha} \varphi_P^{old}
\]

(3.19)

For the dependent variables this approach cannot be followed, since there is no equation to be solved. In that case we follow a similar approach. The new value for the dependent variable is constructed by adding a percentage \( \alpha \) of the difference between the old and the computed new solution \( \varphi_P^{comp} \) to the old solution:

\[
\varphi_P^{new} = \varphi_P^{old} + \alpha (\varphi_P^{comp} - \varphi_P^{old})
\]

(3.20)

Apart from this method additional relaxation was obtained by using small values of \( \Delta t \) in the computations. For stationary flows, as we are computing, this so-called false time step makes the matrix in (3.21) more diagonally dominant. For small enough values of \( \Delta t \), \( \alpha \) might even be set to 1.

### 3.3.3 Convergence criteria

The system of the linearised equations, as described by 3.19, has to be solved for every variable in every outer iteration. We can also write this system in vector notation as follows:

\[
A \varphi = b
\]

(3.21)
where $A$ is the sparse 7-diagonal matrix of size $N_x N_y N_z \times N_x N_y N_z$. The main
diagonal consists of the elements $a_P$, whereas the 6 non-zero off-diagonals store the
values $a_{nb}$. $\varphi$ is the solution vector for one specific independent variable and $b$ the
right-hand side. The procedure to solve this matrix equation is treated in the next
section.

When using iterative methods, one has to establish some acceptable error norm,
below which the computations are stopped and convergence is declared. In our case
inner- and outer iterations are performed, and both should obey prescribed conver-
gence criteria. As stated previously, only a few inner iterations are performed for
every variable, typically 5 to 10 for the pressure and 1 or 2 for all other variables.
When the outer iterations converge, all convergence criteria for the inner iterations
should automatically be satisfied. Nevertheless it is handy to have an error criterion
for the linearised system. It gives a good indication of the state of the outer iterations
and can of course always be used to check whether the inner iterations can be driven
to any accuracy if the user demands so. Two criteria can be applied:

$$R^1(\varphi^n) = \frac{||A\varphi^n - b||}{||\text{diag}(A)\varphi^n||}; \quad R^2(\varphi^n) = \frac{||A\varphi^n - b||}{||b||} \tag{3.22}$$

If the norm of the vector $b$ becomes smaller than machine accuracy (we computed
with 64-bit accuracy, hence machine accuracy is approximately 16 decimals, and thus
the machine accuracy was set to $10^{-16}$), the absolute norm is used: $R^2(\varphi^n) = ||A\varphi^n||$.
Peeters (1995) claims that the first criterion gives a better estimate in case of source
term dominated variables, like turbulent kinetic energy. The right one is more com-
mon among mathematicians. Typical values for $R^1$ and $R^2$ are $10^{-8}$.

More interesting is the determination of convergence criteria for the outer itera-
tions. Two additional criteria were applied. First of all the relative change of all
variables in one representative monitor node of the domain should become small be-
tween successive iterations:

$$R^3(\varphi^n) = \frac{||\varphi_{mon}^n - \varphi_{mon}^{n-1}||}{||\varphi_{mon}^n||} \tag{3.23}$$

The value of $R^3$ was set to $10^{-4}$. The second, and especially in furnace computa-
tions important, criterion concerns the global mass, heat and mixture fraction balances.
The residuals in the balances are obtained by evaluating:

$$R^4(\varphi^n) = \int_{\partial \Omega} \rho \varphi^n u_i n_i dV \tag{3.24}$$

Here $\partial \Omega$ denotes the external boundary of the domain. Putting $\varphi^n$ to 1 yields the
mass balance, while putting $\varphi^n$ to $f$ leads to a mixture fraction balance. This balance
is especially important since this quantity determines the thermochemical variables.
The heat balance is obtained by putting $\varphi^n$ to $h$. Here of course, heat losses to the
wall due to radiation, convection and conduction have to be taken into account. All
these balances should be satisfied within $10^{-2\%}$. Usually this criterion is the most stringent. If this is met, the other criteria are usually satisfied as well. A last criterion is the sum and the maximum of the pressure-correction field. In a converged solution both the sum and the maximum of the pressure-correction should (almost) be zero.

3.4 The iterative solution procedure

3.4.1 Linear system solvers

The matrix equation in equation (3.21) has to be solved. In principle one could invert the matrix $A$ and obtain the solution by directly computing $A^{-1}b$. However, inversion is a very expensive operation, requiring $O((N_xN_yN_z)^3)$ operations. In our case, the matrix $A$ changes every outer iteration for every variable, because of the coupling, which means that this operation has to be performed many times. Also the memory requirements for storing a complete inverse matrix $A^{-1}$ are very high. A second argument is that we are not interested in the exact new solution. Since the equations are linearised and coupled using old values of all variables, the solution to the difference equation that is actually solved is only an approximate solution. This solution will differ per outer iteration, unless all fields would be known exactly, in which case the solution would be converged and no iterations are needed anymore.

Based on above arguments we decided to only use iterative solvers, which are typically one or two orders of magnitude less expensive than direct solution methods, depending on the number of iterations performed and the specific iterative solver. In this study three different solvers have been applied: the space-Tri-Diagonal-Matrix-Algorithm (SP-TDMA, a three dimensional version of the Thomas Algorithm, explained by Post (1988)); the Strongly Implicit Method, SIP, described by Stone (1968) and the Generalised Minimal Residual Algorithm, GMRES as explained by Saad and Schultz (1986)). Especially on Krylov subspace methods there is a plethora of literature. We refer to Van der Vorst et al. (1991).

The performance of any of these solvers depends heavily on the condition number of the matrix. The condition number $\kappa_n(A)$ with respect to the norm $n$ is defined as:

$$\kappa_n(A) \equiv \|A\|_n\|A^{-1}\|_n$$  \hspace{1cm} (3.25)

We have used the $\infty$ norm since this one is easily estimated by LAPACK routines (Anderson et al., 1992). This is obtained by using the $\infty$ matrix norm, defined as the maximum of all rowsums: $\|A\|_\infty \equiv \max_i \sum_j |a_{ij}|$

A condition number gives an indication on the sensitivity of the solution toward perturbations in the right-hand side $b$ or the matrix $A$. The smaller the condition number, the smaller is the influence of perturbations on the solution. This condition number depends, in turn, heavily on the size and stretching of the grid. To show this, we computed the estimated condition number of the matrix describing the two-dimensional Poisson equation $\nabla^2 = b$. The domain was chosen to be a rectangle with
area $L^2$ and two grid spacings were considered; a uniform grid of $N \times N$ cells and a nonuniform grid, where the grid spacing was taken from Henkes (1990).

$$\frac{x_i}{L} = \frac{i}{N} - \frac{\sin(2\pi i/N)}{2\pi}$$ (3.26)

He used this grid to compute natural convection in an enclosure.

The condition numbers $\kappa_\infty(A)$ were estimated from the LAPACK routine DGBCON and are depicted in figure 3.5. Although these systems are still very small considered to real applications, the dependence of the condition number on the grid spacing is clear.

![Figure 3.5: Estimated condition number for two dimensional Poisson equation on uniform (squares) and non-uniform (circles) grids.](image)

### Comparison of linear solvers

Ferziger and Perić (1996) have performed comparisons between line TDMA, SIP and two Krylov subspace methods, ICCG and IC-CGSTAB in combination with multigrid. They conclude that when very accurate solutions are needed, multigrid solvers and ICCG are the best choices. However, if moderate accuracy is needed (as is usually the case in solving non-linear problems), SIP becomes competitive especially in combination with multigrid.

In our study the SP-TDMA has been exploited, rather than the line TDMA version, since the line TDMA shows very poor convergence behaviour in most cases. The SIP solver has been taken from the website ftp.springer.de, and is hence identical to the version used by Ferziger and Perić (1996). The IC-CGSTAB solver has also been taken from there to get a clear comparison between the SP-TDMA solver and the two solvers used by Ferziger and Perić (1996). In later computations CGSTAB
has been replaced by GMRES. Both have a similar convergence behaviour, but GMRES requires less computations per iteration if not too many inner iterations are performed.

To investigate the performance of the different methods we solved the Laplace equation with boundary conditions \( \varphi(0, y, z) = \varphi(x, 0, z) = \varphi(x, y, 0) = 0, \varphi(1, y, z) = yz, \varphi(x, 1, z) = xz, \varphi(x, y, 1) = xy \). The exact solution is known: \( \varphi(x, y, z) = xyz \). This solution is linear in all coordinate directions, so a second order scheme gives the exact solution for any grid gridsize. This makes comparison of the quality of different solvers for this test case very convenient.

This equation was solved on a uniform grid of \( N \times N \times N \) cells, where \( N \) was varied from 16 to 64. Table 3.2 shows the number of iterations and the time needed to reduce the norm of the error by an 4 orders of magnitude with respect to the error in the initial guess. Error criterion \( R^1 \), as defined in equation (3.22) was used. The computations where performed on the CRAY T3E. We see that CGSTAB yields the best results, as also found by Ferziger and Perić (1996). However, more interesting is the performance of the solver in a limited number of iterations. In our calculations a fixed number of inner iterations is usually performed, rather than demanding a specific residual drop, because this makes load balancing easier in parallel computing. This will be elaborated in a chapter 5. Furthermore we already mentioned that the systems we solve for a specific variable \( \varphi \) in a specific outer iteration are the linearised, decoupled approximations of the difference equations, and hence we are not interested in a completely converged solution of the approximated system. The only reason to solve systems very accurately in an inner iteration is that this might be beneficial for the convergence behaviour (this especially holds for the pressure correction, since this serves as a mass deficit equation). Table 3.3 shows the residuals drop in 10 iterations (a realistic value), and how much time the solver takes. The same error criterion was used. Now the SIP solver leads to the best results, as was also concluded by Ferziger and Perić (1996). The SP-TDMA solver performs reasonably well. The CGSTAB method, although leading to the lowest residual and performing remarkably well on the coarse \( 16^3 \) grid, takes almost 2.5 times longer to solve larger systems. From these results it can be concluded that the SIP solver is the most adequate for practical purposes, as it combines speed with good performance.

<table>
<thead>
<tr>
<th>Grid</th>
<th>SP-TDMA</th>
<th>SIP</th>
<th>CGSTAB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>iterations</td>
<td>time</td>
<td>iterations</td>
</tr>
<tr>
<td>( 16^3 )</td>
<td>49</td>
<td>1.11</td>
<td>17</td>
</tr>
<tr>
<td>( 32^3 )</td>
<td>154</td>
<td>37.62</td>
<td>50</td>
</tr>
<tr>
<td>( 64^3 )</td>
<td>426</td>
<td>956.57</td>
<td>151</td>
</tr>
</tbody>
</table>
Table 3.3: Residual obtained and time in seconds required in 10 iterations for the Laplace equation.

<table>
<thead>
<tr>
<th>Grid</th>
<th>SP-TDMA residual</th>
<th>SP-TDMA time</th>
<th>SIP residual</th>
<th>SIP time</th>
<th>CGSTAB residual</th>
<th>CGSTAB time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16^3$</td>
<td>$2.597 \times 10^{-2}$</td>
<td>0.238</td>
<td>$8.249 \times 10^{-4}$</td>
<td>1.159</td>
<td>$3.970 \times 10^{-5}$</td>
<td>0.520</td>
</tr>
<tr>
<td>$16^3$</td>
<td>$8.331 \times 10^{-2}$</td>
<td>2.622</td>
<td>$2.758 \times 10^{-2}$</td>
<td>2.816</td>
<td>$1.752 \times 10^{-2}$</td>
<td>5.190</td>
</tr>
<tr>
<td>$64^3$</td>
<td>$1.305 \times 10^{-1}$</td>
<td>24.33</td>
<td>$8.442 \times 10^{-2}$</td>
<td>18.62</td>
<td>$7.013 \times 10^{-2}$</td>
<td>44.45</td>
</tr>
</tbody>
</table>

As mentioned before however, the behaviour of the solver depends strongly on the quality of the grid and the size of the grid. Therefore the SP-TDMA, SIP and GMRES solver were tested on a turbulent combustion simulation in a furnace. It is very interesting to see if the above conclusions, drawn for simple geometries and uniform grids can be extrapolated to more complex geometries and very stretched grids. A stretched grid of size $16 \times 24 \times 20$ CVs was used, covering a domain of $0.44 \text{ m} \times 4.09 \text{ m} \times 0.955 \text{ m}$. A detailed description of these furnace computations is given in section 3.6. The grid used in these solver studies is shown in figure 3.6. Note the high aspect ratio's occurring near the end of the furnace. They are a result of the required grid size near the gas inlet. An intermediate solution (i.e. not completely converged) was chosen as a starting point for all computations. 100 outer iterations were performed, using respectively 1 and 10 iterations in all the solvers. Table 3.4 shows the predicted

Figure 3.6: Coarsest (16-24-20) grid for furnace computations, shown together with furnace geometry
3.4. The iterative solution procedure

global mass imbalance after 100 iterations. This mass imbalance was obtained by taking the norm of the source term for the pressure correction equation, which is the sum of the mass fluxes through all CVs. The SIP solver again performs best. Note

<table>
<thead>
<tr>
<th>Number of iterations</th>
<th>SP-TDMA</th>
<th>SIP</th>
<th>GMRES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.35 10^{-4}</td>
<td>2.95 10^{-4}</td>
<td>1.25 10^{-3}</td>
</tr>
</tbody>
</table>
| 10                   | 2.86 10^{-4}| 2.63 10^{-4}| 4.34 10^{-4} | (*)

(*) No restarts performed for GMRES.

the poor performance of the GMRES solver. It starts off very slowly, leading to a very high mass imbalance after one iteration. When increasing the number of iterations, the improvement of the GMRES solver is higher than using the other solvers. This may have to do with the fact that the grid for this computation contains cells with very large aspect ratios, which, as mentioned, increases the condition number of the matrix. The SIP and SP-TDMA solver seem to suffer less from this grid layout.

The same conclusion can be drawn for all other computed variables. Table 3.5 shows the residuals, according to criterion 3.22 for all computed variables. However,

<table>
<thead>
<tr>
<th>Variable</th>
<th>Inner iter.</th>
<th>SP-TDMA</th>
<th>SIP</th>
<th>GMRES</th>
</tr>
</thead>
<tbody>
<tr>
<td>U-velocity</td>
<td>1</td>
<td>1.076 10^{-05}</td>
<td>3.614 10^{-06}</td>
<td>2.741 10^{-03}</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>7.113 10^{-15}</td>
<td>3.363 10^{-16}</td>
<td>5.632 10^{-04}</td>
</tr>
<tr>
<td>V-velocity</td>
<td>1</td>
<td>1.638 10^{-06}</td>
<td>3.106 10^{-07}</td>
<td>5.215 10^{-04}</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2.255 10^{-15}</td>
<td>1.098 10^{-16}</td>
<td>6.551 10^{-05}</td>
</tr>
<tr>
<td>W-velocity</td>
<td>1</td>
<td>2.208 10^{-06}</td>
<td>6.165 10^{-07}</td>
<td>1.318 10^{-03}</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3.265 10^{-15}</td>
<td>1.091 10^{-16}</td>
<td>1.349 10^{-04}</td>
</tr>
<tr>
<td>Turbulent kinetic energy</td>
<td>1</td>
<td>6.071 10^{-07}</td>
<td>1.780 10^{-07}</td>
<td>2.100 10^{-03}</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>7.602 10^{-17}</td>
<td>4.407 10^{-15}</td>
<td>3.114 10^{-05}</td>
</tr>
<tr>
<td>Turbulent dissipation</td>
<td>1</td>
<td>1.109 10^{-09}</td>
<td>1.088 10^{-09}</td>
<td>3.467 10^{-05}</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>8.559 10^{-17}</td>
<td>8.195 10^{-16}</td>
<td>1.701 10^{-08}</td>
</tr>
<tr>
<td>Mean mixture fraction</td>
<td>1</td>
<td>4.041 10^{-12}</td>
<td>1.732 10^{-13}</td>
<td>8.513 10^{-08}</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>3.974 10^{-30}</td>
<td>3.717 10^{-31}</td>
<td>6.217 10^{-10}</td>
</tr>
<tr>
<td>f-variance</td>
<td>1</td>
<td>1.219 10^{-16}</td>
<td>2.771 10^{-17}</td>
<td>4.959 10^{-10}</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1.743 10^{-33}</td>
<td>2.386 10^{-31}</td>
<td>2.453 10^{-14}</td>
</tr>
<tr>
<td>Mean enthalpy</td>
<td>1</td>
<td>7.424 10^{-07}</td>
<td>1.482 10^{-07}</td>
<td>2.940 10^{-04}</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1.431 10^{-15}</td>
<td>1.167 10^{-16}</td>
<td>3.217 10^{-05}</td>
</tr>
</tbody>
</table>
the performance of solvers remains very problem dependent, and especially in sim-
ulations of complete furnaces in complex geometries, the quality of a solver is often
lost in very nonlinearly stretched grids and the fact that the coupling of the systems
leads to very stiff systems, further increasing the condition number of the matrix.
All this makes a decent comparison of the quality of solvers in turbulent combustion
simulations very difficult. Consequently it was decided to use the SIP solver in the
all the computations for all variables.

3.4.2 Multigrid

The above results show that all iterative methods discussed converge more slowly on
finer grids. Both the outer- and inner iterations suffer from the increase in grid cells.
The amount of inner iterations increases because the condition number of the matrix
increases. This leads to very slow damping of the error components of the system.

But even if the linear systems would be solved directly, the number of outer
iterations would still increase, due to the iterative nature of equation coupling and
the treatment of non-linearities. According to Ferziger and Perić (1996), page 297
"This behaviour is related to the fact that information travels only one grid [cell]
per direction and, for convergence, information has to travel back a forth across the
domain several times." Muzafesrija and Gosman (1997b) even states that the number
of outer iterations increases linearly with increasing number of CVs.

Multigrid yields a very effective way to improve convergence of both the inner- and
outer iterations. Since we are not really interested in solving the systems exactly, we
have only applied multigrid to the outer iterations. However, Wesseling (1992) gives
a good review of applying multigrid to inner iterations. This leads to a very robust
method, which is also less sensitive to the variation of underrelaxation factors than
the single-grid version. The multigrid method we used is called Full Multi Grid, and
details of the method can be found in Muzafesrija and Gosman (1997b) and Ferziger

With multigrid, the solution is first obtained on a coarse grid for all variables.
It is assumed that the grid is coarse enough to compute a converged solution easily.
This grid will often be too coarse to render grid independent solutions. Hence in the
second step of the multigrid method a solution on a finer grid is sought. The finer grid
is obtained by dividing every coarse grid cell in eight fine grid cells. This is depicted
in figure 3.7 for the two dimensional case. Since we use Cartesian grids, the location
of the new cell faces and nodes can be easily determined. The coarse (level 1) grid
solution serves as a good initial starting guess for the fine (level 2) grid solution. As
the multi-grid tends to be slow in the beginning of the iteration, it is important to
start with a 'good approximation'. Therefore, a solution of this coarser grid can be
used. This approach is called the Full Multi-Grid method (FMG) (Wesseling, 1992).

On two grid levels the method becomes as follows. First the fine grid solution is
computed using the values obtained from the coarse grid simulation. We can write:

\[ A_h \varphi_h = B_h \]  

(3.27)

analogous to (3.21), but now the matrix \( A \) is the operator, consisting of all variables and \( \varphi \) contains all unknowns in all grid points. The subscript \( h \) denotes the grid spacing. After solving 3.27 to a certain accuracy, the equation is satisfied up to a residual \( R_h \):

\[ A_h \varphi_h - B_h = R_h \]  

(3.28)

This solution is now interpolated (restricted) to the coarse grid with spacing \( 2h \). The equations to be solved on the coarse grid are not the equations one would solve if that grid were used by itself; they are rather a 'smoothed' version of the fine grid solution. This is because we are now solving for approximations of the fine grid solution, rather than for the solution itself on the coarse grid level. This is why this type of multigrid methods is called the Full Approximation Scheme. There is an additional source term containing the restricted fine grid equation, and, due to the non-linearity of the equations, a slight change in the coefficient matrix. The equation on the coarse grid can be written as:

\[ \tilde{A}_{2h} \tilde{\varphi}_{2h} - \tilde{B}_{2h} = \tilde{A}_{2h} \tilde{\varphi}_{2h} - \tilde{B}_{2h} - \tilde{R}_{2h} \]  

(3.29)
The right-hand side contains the restricted quantities from the fine grid, denoted by $\hat{\cdot}$ and the $\check{\cdot}$ denotes the approximation to the fine grid solution. When the fine grid residual equals zero, no correction is necessary. In this case $R_{2h} = 0$, and consequently $\bar{\varphi}_{2h} = \check{\varphi}_{2h}$. In practise however, the restricted residual is non-zero, leading to the additional source term on the right hand side. After solving the system, the new approximation will deviate from its old value, leading to a correction field $\varphi'_{2h} = \check{\varphi}_{2h} - \bar{\varphi}_{2h}$. This correction is transferred to the fine grid by interpolation (prolongation) and added to the old solution on that level $\varphi_h$. This procedure is repeated until the residual on the fine grid is below some required level. Then a three-grid procedure can be adapted. We used a V-cycle to run over the different grid levels. Tri-linear interpolation is used for restriction and prolongation operators. For further details on the implementation we refer to Twerda and Verweij (1997).

**Performance of FMG**

The performance of the multigrid approach was tested on different test cases (Twerda et al., 1998). First we took a 3D laminar lid-driven cavity flow. Details of this test case are explained in section 3.6. Uniform grids of $N \times N \times N$ cells were applied, where $N$ was varied between 4 and 64. Multigrid was performed with the $4^3$ grid as coarsest level. Hence for the $N=8$ case a maximum of two levels of multigrid were applied, whereas for the $N=64$ case up to five levels of multigrid could be applied. Table 3.6 shows the convergence history for several grid sizes and several multigrid levels for $Re = 1$ and $Re = 100$. All tests have been performed on a HP 9000/770.

<table>
<thead>
<tr>
<th>Re</th>
<th>grid size</th>
<th>Iterations</th>
<th>CPU times</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Multigrid levels</td>
<td>Multigrid levels</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 2 3 4 5</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>1</td>
<td>$4^3$</td>
<td>44 - - - -</td>
<td>- - - -</td>
</tr>
<tr>
<td></td>
<td>$8^3$</td>
<td>- 44 - - -</td>
<td>- - - -</td>
</tr>
<tr>
<td></td>
<td>$16^3$</td>
<td>- - 34 - -</td>
<td>- - - -</td>
</tr>
<tr>
<td></td>
<td>$32^3$</td>
<td>- - - 47 -</td>
<td>- - - -</td>
</tr>
<tr>
<td></td>
<td>$64^3$</td>
<td>- - - - 50</td>
<td>- - - -</td>
</tr>
<tr>
<td>100</td>
<td>$4^3$</td>
<td>100 - - - -</td>
<td>153 - - - -</td>
</tr>
<tr>
<td></td>
<td>$8^3$</td>
<td>110 54 - - -</td>
<td>172 327 - - -</td>
</tr>
<tr>
<td></td>
<td>$16^3$</td>
<td>373 84 39 - -</td>
<td>653 534 418 - -</td>
</tr>
<tr>
<td></td>
<td>$32^3$</td>
<td>1142 249 74 45 -</td>
<td>3478 2474 1088 862 -</td>
</tr>
<tr>
<td></td>
<td>$64^3$</td>
<td>- - 223 64 40</td>
<td>- - 9667 3056 2162</td>
</tr>
</tbody>
</table>
series workstation. These results are in agreement with Ferziger and Perić (1996). The number of outer iterations becomes almost independent of the gridsize, which was the main reason for applying the multigrid approach. This test case might, however, not be representative for the performance in more complex flows. Therefore, the FMG was applied to the turbulent backstep flow and the simulation of the IFRF furnace. Details of these cases are given in section 3.6.

In the backstep flow computation four levels of multigrid were used. The coarsest grid had $6 \times 24$ CVs, uniformly divided. The single grid approach is denoted by SG. Figure 3.8 shows the convergence history for the $k$ equation.

![Convergence history graph](image)

**Figure 3.8**: Multigrid behaviour for the $k$-equation in a turbulent backstep flow simulation

The convergence behaviour for the FMG approach is again orders of magnitudes better than for the SG method and almost independent of the number of CVs. This is in agreement with other results Lilek and Perić (1995). Note also that the rate of convergence does not deteriorate.

Finally the FMG approach was applied to the IFRF furnace simulation. The coarsest grid consist of $12 \times 24 \times 14$ CVs. The number of grid points for the coarsest grid could not be further reduced due to geometrical restrictions. At least 1 CV should be used to cover the fuel inlet, which is only 6 mm in diameter. Furthermore, the relatively thin flame front would not be correctly captured by an even coarser grid.
Two levels of multigrid were used. The convergence behaviour for the mean mixture fraction is shown in figure 3.9. In this case the FMG was used in conjunction with

![Graph showing convergence behaviour]

**Figure 3.9:** Convergence behaviour of the mean mixture fraction in a furnace simulation when applied with a single grid (SG) and two levels of multigrid (FMG).

the domain decomposition, which will be discussed in chapter 4. Calculations were done using 16 blocks. Twerda et al. (1998) discusses the influence of the domain decomposition on the multigrid behaviour. Also Baysal et al. (1991) have performed multigrid computations combined with the embedded grid approach.

It can been seen that the multi-block MG converges much better than the multi-block SG solver, but stagnates after 1800 iterations. This is probably due to the insufficient description of the curved walls by the porosity method. In our study the complex geometry is modelled with the porosity method (Moult et al., 1979), which marks the cells that lie outside the fluid domain in the Cartesian mesh as 'closed cells'. In these cells all quantities are set to a preset value, and no equation is solved for these nodes. The equations for the CVs that partly fall outside the domain are modified for the fact that part of the CV is blocked by the wall. The porosity method has some difficulties to model the curved walls consistently on different grid levels. If more complex geometries are needed, more attention must be given to this problem.
3.5 Radiation solution method

The radiative heat transfer equation, as given by equation 2.28, differs from all other equations to be solved in the sense that it is an integral equation rather than a convection-diffusion equation. Even for non-scattering media exact solutions to this equation have only been found for very simple geometries. The integration over the solid angle still requires numerical approximation methods. In this study the Discrete Transfer Method (DTM) was applied to solve equation (2.28), described in detail by Lockwood and Shah (1981) and Wieringa (1992), who applied the DTM to furnace computations.

The DTM starts by dividing the geometry into control volumes, just as with the Finite Volume Method. In each boundary surface element \( A_j \) the hemisphere is divided into a finite number of solid angle elements, with the centre of the surface element \( P \) as origin. For every solid angle element a beam direction is defined, in which the directional incident flux is calculated. This is depicted in figure 3.10. Each

![Figure 3.10: Division of geometry into DTM-cells.](image)

beam is considered to be an approximation to the total radiative flux of the entire solid angle element. The beam impinges on the surface in centre point \( P \) by definition. The direction of the beam is given by \( \theta_k \), allowing for the point \( Q_k \) from where it has emerged to be calculated, by tracing the beam towards the other side of the enclosure. The beam is given an energy dependent on the boundary conditions and followed as it crosses the various DTM control volumes on its travel from \( Q_k \) to \( P \) in the points \( n-1, n \) etc. At the faces of each control volume the intensity is determined, until the beam has reached the first surface element, point \( P \), where now the incoming intensity from this solid angle element is known.

The DTM has the advantage that the absorption-emission coefficient can be non-constant and position dependent without increasing the computational effort. How-
ever, drawbacks are the 'ray-effect' (the dependency of the accuracy of the method on the number of beams reaching point $P$) and energy conservation is not always satisfied. Wieringa (1992) pays much attention to the accuracy of the method and formulates additional criteria for the beam directions to satisfy energy conservation.

**Grey radiation**

The derivation of a numerical approximation method starts with discretising the radiative transfer equation (2.28). It reads:

$$i'_{\lambda,n} = i'_{\lambda,0} \tau_{\lambda}(0,n) + \sum_{j=1}^{n} i'_{b,\lambda,j}(\tau_{\lambda}(j,n) - \tau_{\lambda}(j-1,n))$$  \hspace{1cm} (3.30)

Here the subscript $n$ has been defined in figure 3.10. If the gas is grey, the dependence on the wavelength can be dropped, and equation 3.30 can be simplified to the following recurrence relation:

$$i'_n = i'_{n-1} e^{-\kappa \delta} + i'_b(1 - e^{-\kappa \delta})$$  \hspace{1cm} (3.31)

Here $\kappa$ is the grey absorption-emission coefficient in the volume between $(n-1)$ and $n$, see figure 3.10 for the other quantities. The radiant intensity leaving from $Q_k$ is:

$$i'_0 = q_0^{-1}/\pi$$  \hspace{1cm} (3.32)

where it is assumed that the surface element is diffuse and isothermal, and that the flux $q_0^-$ at the boundary $Q_k$ is known. After solving this recurrence relation, the total hemispherically incident flux on surface $A_j$ can be found by summing the intensities from all directions $k$:

$$q_j^+ = \sum_k i'_k \cos \theta_k \Delta \Omega_k$$  \hspace{1cm} (3.33)

The total radiative source of volume $m$, $S_{rad,m}$ for a grey gas is finally found by summing over all beams that cross that volume:

$$S_{rad,m} = \frac{1}{V_m} \sum_k (i'_{n-1} - i'_n) A_k \cos \theta_k \Delta \Omega_k$$  \hspace{1cm} (3.34)

with $V_m$ the volume of the DTM control volume. The beams are travelling towards surfaces $A_k$ and make an angle $\theta_k$ with the normal of these surfaces.

**Spectral radiation**

If the gas is not assumed to be grey, the simplification 3.31 does not hold, and equation 3.30 has to be used. In principle this equation provides the solution for the calculation of spectral radiative heat transfer, once temperatures and gas composition are
3.5. Radiation solution method

known. However, as stated in the previous chapter, it is impossible to compute the transmissivity in such narrow wavelengths that $\kappa_\lambda$ can be assumed constant. An average transmissivity $\tau_\lambda \equiv 1/\Delta \lambda \int_{\Delta \lambda} \tau_\lambda \, d\lambda$ is used instead. The total spectral radiative source term for control volume $m$ can then be approximated, following a similar reasoning as above, by:

$$S_{\text{rad},m} = \frac{1}{V_m} \sum_k \sum_{\lambda} (i'_{\lambda,n-1} - i'_{\lambda,n}) \Delta \lambda A_k \cos \theta_k \Delta \Omega_k$$

(3.35)

Note that this is the discretised version of equation 2.31.

Mapping the radiation grid onto the FV grid

The computational effort for the DTM is proportional to $N_{\text{rad}}^3 N_{\text{beam}}$ (grey gas calculations) or $N_{\lambda}^3 N_{\text{beam}} N_{\lambda}$ (spectral calculations). Here $N_{\text{rad}}$ is the number of radiation volumes in one direction, $N_{\text{beam}}$ is the number of beams that leaves each surface and $N_{\lambda}$ the number of wavelength intervals.

We could, in principle, use the finite volume (FV) grid also for the DTM calculations, but some severe problems arise when a very fine DTM grid is applied; apart from a huge computational effort (the radiative heat transfer equation is an integral equation for which the domain decomposition approach, explained in next chapter, does not hold), the accuracy of the DTM becomes much more sensitive to the distribution of the rays. Boerstoel (1997) discusses these problems and suggests to use a DTM grid that is coarser than the FV grid, which we did.

The inherent problem to be solved consequently is the mapping of the required values in the DTM cells from the values in the FV, and the mapping of the computed values in the DTM cells to the values in the FV cells. We use the method suggested by Kuyper (1994): The black body emissive power and absorption coefficient for radiation volume $k$ are taken to be volume weighted averages over all FV cells (with index $i$) that lie completely or partly within volume $k$. We obtain:

$$
e_{b,k} = \frac{\sum_i w_i v_i e_{b,i}}{V_k} ; \quad \kappa_k = \frac{\sum_i w_i v_i \kappa_i}{V_k}$$

(3.36)

where the summation is over all FV cells, $v_i$ is the volume of a FV cell, $V_k$ the volume of the radiation cell and $w_i$ a weighting factor which is equal to the fraction of the FV cell within radiation volume $k$. So $w_i$ is 1 when the cell lies entirely within the radiation volume and 0 for a cell entirely out of the radiation volume. Gas concentrations are averaged similarly.

After computing the radiative source term $Q_{\text{rad}}$ on the DTM grid, the term is split into an absorption part and an emission part:

$$Q_{\text{rad}} = Q_{\text{rad,abs}} - Q_{\text{rad,em}}$$

(3.37)

These two contributions have to be distributed differently. Radiative emission is proportional to $v$, $e_b$ and $\kappa$; radiative absorption is proportional only to $v$ and $\kappa$. So
we obtain for FV cell P which lies within radiation volume $k$ the following radiative source:

$$Q_{\text{rad},P} = Q_{\text{rad,abs}} \times \frac{w_{P \nu k P}}{V_k \kappa_k} \times \frac{w_{P \nu \epsilon_{b,k} P K_P}}{V_k \epsilon_{b,k} \kappa_k}. \quad (3.38)$$

Boerstoel (1997) formulates the criteria to be satisfied when applying the DTM: First of all the size of the radiation volumes must be such that: $L_m \kappa < 0.15$ where $\kappa$ is the average absorption-emission coefficient and $L_m$ the mean beam length of this volume. Secondly, the difference of the temperature of the incoming radiation and the temperature within the radiation volume may not be very large (less than 5%).

### 3.6 Description of computed test cases

Finally we will give an overview of the test cases mentioned in this thesis. We have taken four test cases to see the importance of different simulations on block decomposition: A convection-diffusion equation, a two- and three-dimensional laminar lid-driven cavity flow, a two-dimensional turbulent backstep flow and finally the complete turbulent combustion in the so-called IFRF furnace. All of these cases have been computed with the same code, and hence serve as a good benchmark set to check the validity of the code. Also new code patches can be checked on correct implementations, by checking if the results of the new version of the code are identical to the results obtained with the old version of the code (or at least only expected differences occur.)

### Convection diffusion equation

The simplest test case for a CFD simulation algorithm is a convection diffusion equation describing the transport of a scalar quantity using a given velocity field. This one has been taken from Carvalho et al. (1991). The equation being solved was:

$$\frac{\partial}{\partial x}(U \varphi) + \frac{\partial}{\partial y}(V \varphi) = \frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} + 2\pi^2 \varphi \quad (3.39)$$

$$U = -\lambda \pi \sin(\pi x) \sin(\pi y)$$

$$V = -\lambda \pi \cos(\pi x) \cos(\pi y)$$

On the domain $x \in [0, 1]$, $y \in [-0.5, 0.5]$. The analytical solution to this equation exists (Domingos and Lopes, 1981) and is given by:

$$\varphi = \sin(\pi x) \cos(\pi y) \quad (3.40)$$

This analytical solution is plotted in figure 3.11.
3.6. Description of computed test cases

\[ \sin(\pi x) \cdot \cos(\pi y) \]

\[ \phi (x,y) \]

Figure 3.11: Exact solution for the convection diffusion equation, described by Carvalho (1991).

Laminar flow

The next test case involves solving the velocity field, rather then assuming it known. The simplest simulations of this kind involve laminar flows. A three-dimensional lid-driven cavity flow was used. Two versions for the laminar lid-driven cavity flow have been used. One described by Shih and Tan (1989) and one described by Thompson and Ferziger (1989). Both cases are two-dimensional test cases. To use these cases as a three-dimensional method, one can either apply symmetry conditions in the third direction (in which case the stream function can be computed, as there is no dependency of the third dimension in the solution) or apply solid-wall boundary conditions. This has been done by Muzafferija and Gosman (1997b). In this case the proper implementation of the boundary condition, and the symmetry of the solution can be checked. The symmetry conditions have been applied in our case, unless stated otherwise.

The case proposed by Shih and Tan (1989) includes a quite complicated source term included and a quadratical velocity distribution on the top lid. This quadratic
lid velocity \((U_{lid} = 16x^2(x - 1)^2, x = 0, 1), x \in [0, 1]\) has the advantage that the flow velocity at the two corners is zero, removing the ambiguity of the velocity at these points when specifying a constant lid velocity. For this benchmark problem an exact analytical solution is available. Density and viscosity were set to 1. The Reynolds number based on the maximum lid velocity \((U_{lid}(x = 0.5) = 1)\) and the length \(H\) of the cavity is \(Re=1\). The pressure field and streamlines for this case are shown in figure 3.12. Note that there is no recirculation in the corners. The reason that

![Figure 3.12: Pressure field and velocity field for the laminar lid-driven cavity flow at Re= 1, as proposed by Shih (1989). Isolines for pressure levels -0.7, -0.5, -0.3, -0.1, 0.0, 0.1, 1, 2, 3, 4, 5. Isolines for streamlines -0.125(0.06)0.](image)

this testcase was not used for all laminar simulations is that the cavity flow with the uniform lid velocity \(U_{lid}\) is much more common as a benchmark. This variant was also used by Thompson and Ferziger (1989), and values for the streamfunction are known accurately (Ghia and Shin, 1982). The density and uniform lid velocity \(U_{lid}\) were set to 1. Reynolds numbers \(Re = U_{lid}H/\nu = 1\) and 100 are computed. Figure 3.13 shows the profile of the \(U\) velocity in the middle of the cavity for \(Re=100\) for uniform grids of \(N \times N \times N\) CVs, with \(N\) ranging from 4 to 64. The central differencing scheme was applied for the convective terms. As can be seen, the solutions on the two finest grids do not differ much. To check the quality of the solution, the minimum value of the streamfunction was also evaluated. Table 3.7 shows the minimum value of the streamfunction.
3.6. Description of computed test cases

![Graph showing u-velocity profile at plane z=0.5 for the laminar lid-driven cavity flow, as proposed by Thompson (1989)]

**Figure 3.13:** u-velocity profile at plane z=0.5 for the laminar lid-driven cavity flow, as proposed by Thompson (1989)

**Table 3.7:** Minimum value of the streamfunction for Re= 100

<table>
<thead>
<tr>
<th>grid size</th>
<th>$4^3$</th>
<th>$8^3$</th>
<th>$16^3$</th>
<th>$32^3$</th>
<th>$64^3$</th>
<th>Ghia(1982)</th>
</tr>
</thead>
<tbody>
<tr>
<td>streamfunction</td>
<td>-0.049</td>
<td>-0.079</td>
<td>-0.095</td>
<td>-0.101</td>
<td>-0.103</td>
<td>-0.103</td>
</tr>
</tbody>
</table>

**Turbulent flow**

The following test case is the two dimensional turbulent backstep flow. This flow is one of the simplest turbulent flows, because of the known location of the separation point, the straightforward geometry and the plethora of literature on the topic, and still one of the most difficult to compute for standard turbulence models, because of the adverse pressure gradient and the secondary eddy immediately after the backstep. The present testcase stems from Le et al. (1993). The Reynolds number based on the step height and mean free-stream velocity is 5100. The grid used for all computations is shown in figure 3.14. The boundary conditions at the inlet were taken from the DNS data (Le et al., 1993), available in the ERCOFTAC database. Figure 3.15 shows
the predicted skin friction coefficient (defined as $2\tau_w/\rho U^2$) along the step, compared to the DNS-data by Le et al. (1993) and results from Jakirlić (1997). Good agreement with similar models is obtained, and also is shown the poor comparison with DNS. These computations show that the implementation of the turbulence model and wall functions seem to be correct. It can be seen that the $k-\varepsilon$ model is not capable of capturing the secondary eddy in the corner.

**Turbulent reacting flows, the IFRF furnace simulation**

The most complex case we have computed involves the simulation of turbulent combustion in a glass melting furnace. The configuration chosen here is the combustion of natural gas in the IFRF test-furnace. This semi-industrial furnace resembles one
part of a high-temperature regenerative glass melting furnace. Figure 3.16 sketches the geometry of the furnace.

![Sketch of the IFRF furnace](image)

**Figure 3.16:** Sketch of the IFRF furnace

The overall dimensions are 0.96 m × 3.75 m × 0.88 m (width × length × height). This is to industrial standards still a small furnace. The diameter of the gas inlet (6 mm) is a few orders of magnitude smaller than all other sizes of the furnace; it is in fact almost invisible in the figure. The gas enters the furnace here with very high velocity, leading to a good turbulent mixing, but giving rise to very steep gradients in the vicinity of the gas inlet.

This case was chosen because the IFRF have performed a series of trials on this specific furnace geometry in October 1990. These so-called NG7-trials are described by Nakamura et al. (1991), Nakamura et al. (1993). The trials were primarily initiated to study NO<sub>x</sub> reduction. They employed a wide range of firing modes, varying the gas inlet location (overport -OP-, underport -UP-), the gas inlet angle α<sub>gas</sub> (6° ... 20° UP, -20° ... -50° OP) and the gas injection velocity (v<sub>gas</sub> = 75 ... 400 m/s). The main operation conditions can be found in above references.

For all cases flue gas composition and temperature, total heat fluxes and wall- roof- and load-temperatures were measured. Two flames where selected for more extensive measurements (in-flame temperatures and concentrations, as well as spectral fluxes), namely the UP firing mode at α<sub>gas</sub> = 16°, v<sub>gas</sub> = 125 m/s and the OP firing mode at α<sub>gas</sub> = -20°, v<sub>gas</sub> = 125 m/s.

Underport firing is most commonly used in glass melting furnaces. Hence it was decided to focus on the UP firing mode at α<sub>gas</sub> = 16°, v<sub>gas</sub> = 125 m/s only. To get some insight in this experiment, some important measured data for this configuration are given in table 3.8. For the computations the upstream parts of the gas-inlet and the air-inlet pipe are not taken into account. Instead, a higher turbulence intensity level, taken from the experiments, is specified. For details we refer to Koster (1993). Furthermore the IFRF furnace is assumed to be symmetrical. Because of this symmetry only half of the combustion chamber has to be simulated. The geometry used in the code is visualised in figure 5.1. Chapter 6 gives predictions for various fields.
Table 3.8: Characteristics of measured UP-flames of the NG7-trials: variation of the gas injection angle for $v_{gas}$ is 125 m/s.

<table>
<thead>
<tr>
<th>$v_{gas}$</th>
<th>$\alpha_{gas}$</th>
<th>6°</th>
<th>12°</th>
<th>16°</th>
<th>20°</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test number</td>
<td>019</td>
<td>020</td>
<td>022</td>
<td>024</td>
<td></td>
</tr>
<tr>
<td>125 m/s</td>
<td>$O_{2,flue}$ (%)</td>
<td>2.0</td>
<td>2.2</td>
<td>2.0</td>
<td>2.1</td>
</tr>
<tr>
<td></td>
<td>$CO_{flue}$ (ppm)</td>
<td>463</td>
<td>115</td>
<td>68</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>$T_{flue}$ (K)</td>
<td>1718</td>
<td>1738</td>
<td>1701</td>
<td>1693</td>
</tr>
<tr>
<td></td>
<td>$T_{roof,av}$ (K)</td>
<td>1463</td>
<td>1494</td>
<td>1505</td>
<td>1522</td>
</tr>
</tbody>
</table>

Detailed comparisons of numerical simulations of this furnace (in particular the two flames mentioned above) and measurements have been carried out by Koster (1993) and Boerstoel (1997). Boerstoel (1997) concludes that average temperatures and heat fluxes are predicted with confidence for all furnace conditions that are studied. Soot is shown to play an important role for the slowly mixing, low velocity flames. When soot is not accounted for, some trends can not be predicted correctly. Predicted in-flame temperatures are, generally, somewhat higher than the measured temperatures but the shape of the profiles is predicted very well and the overall agreement between simulation and experiment is very good. However, for lower gas inlet velocities, the mixing rate is overpredicted resulting in too high temperatures. This might be contributed to errors made by the $k-\varepsilon$ turbulence model in accurately predicting the mixing rate. Improvement of the present simulation model should be sought in a more elaborate turbulence model, and, locally, very fine grids to better resolve the turbulent structures that determine the mixing rate. Several grid sizes were used to perform the computations on this furnace. They are quoted in table 3.9. The $x$-direction specifies the width of the furnace, $y$ specifies the length and $z$ the height.

Table 3.9: Global mesh sizes used in present study

<table>
<thead>
<tr>
<th>Mesh</th>
<th>$N_x \times N_y \times N_z$</th>
<th>#CVs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse</td>
<td>$16 \times 24 \times 20$</td>
<td>7680</td>
</tr>
<tr>
<td>Medium</td>
<td>$32 \times 50 \times 50$</td>
<td>80000</td>
</tr>
<tr>
<td>Fine</td>
<td>$62 \times 98 \times 82$</td>
<td>498232</td>
</tr>
<tr>
<td>Very fine</td>
<td>$97 \times 202 \times 100$</td>
<td>1959400</td>
</tr>
</tbody>
</table>
4 Domain decomposition

4.1 Introduction

In this chapter a general domain decomposition technique on block-structured grids is given. Both staggered and colocated approaches can be handled. The blocks can be coupled in an arbitrary manner. Since the domain decomposition is merely a mathematical manner to change the solution method, the final converged multi-block solution does not differ from the single-block solution.

The effect of block decomposition on the convergence is addressed. The use of complex solvers on structured grids results in a strong coupling between the nodes. Since the domain decomposition breaks this coupling in parts of the domain, convergence loss will occur. We propose an easy-to-implement technique to improve the convergence of an implicit multi-block solver. The improvement is very stable and works as expected on all considered cases. Finally local grid refinement is discussed. Several refinement criteria have been tested to account for adaptive grid refinement.

First we will give the motivations for using domain decomposition. Then we will discuss the approach used in this study and show some results of the domain decomposition on the convergence of the solution in several relevant testcases. The chapter ends with the discussion on the local grid refinement criteria.

4.2 Reasons for using domain decomposition

Domain decomposition is the mathematical equivalent of 'divide and conquer'. Rather than solving the entire problem at once (which involves solving large systems on a fine mesh), the problem is split in smaller, similar, subproblems which can be solved with less effort. The concept of domain decomposition is not new. Schwartz (1869) already addressed the topic, by proving the existence of harmonic functions on irregular regions which are the union of overlapping sub-regions.

In our case the entire problem is split into subproblems by decomposing the entire (global) spatial domain into a number of subdomains and solving the systems on the subdomains. It is the task of the domain decomposition algorithm to couple the
solutions of the subdomains in a correct way, such that the converged solution of the union of subdomains equals the converged solution of the entire problem.

There are several reasons for using a block-structured block decomposition approach.

Local grid refinement

One of the major problems when using structured Cartesian grids, as we do, is that these grids severely restrict the choice in distribution of grid points. Especially in furnace simulations, with a broad range of length scales, where a large domain has to be covered with a fine enough mesh to capture the occurring small-scale phenomena, a single-block grid leads to vast amounts of (mostly inefficiently located) grid points. Using block-structured grids solves the problem, since the grid in every block might be chosen independently from the grid in other blocks. Section 4.7 discusses local grid refinement.

Parallelisation

An important reason for the increased popularity of the domain decomposition approach over the last 10 years is that this method leads to a very natural division of the computations in a simulation. All the subproblems can consequently be handled by different processors. Parallelisation of the domain decomposition code is the topic of chapter 5. Domain decomposition is by far the most employed method to extend CFD codes to parallel platforms (Sciano et al., 1997). A useful introduction to parallelisation of CFD codes is given by Wesseling (1996).

Other reasons

It may be clear that domain decomposition enables the use of much larger grids, without memory- or computing time problems. If a grid is too large to fit into a single processor memory, it is simply divided into a number of smaller grids and divided over several processors. An additional advantage is that the turn-around time of an application (the turn-around or wall-clock time is the time that a computer needs to complete a simulation) does not necessarily increase when increasing the grid size; computing a solution on a coarse grid and computing a solution on a fine grid with twice as many grid points on twice as many processors could, ideally, be done in the same time. Complex geometries, that cannot be mapped easily onto a single rectangular block, can be handled more efficiently using domain decomposition, by dividing the complex geometry into a number of subdomains that can be mapped onto a number of blocks.

Finally, one could even take the idea of employing different grids in different blocks one step further, by also employing different models in different blocks, marking areas
of flame front, high shear and near solid walls for instance. All these possibilities have not been exploited further in this study.

4.3 Basic domain decomposition algorithm

Domain decomposition aims at solving the second-order differential equation

\[ L\varphi = f, \text{ on } \Omega \]  

(4.1)

where \( \Omega \) is the global domain, with suitable boundary conditions on the global boundary \( \Gamma \), by decomposing the domain into \( N \) subdomains (blocks) \( \Omega_i, i = 1, \ldots, N \), with \( \Omega = \Omega_1 \cup \ldots \cup \Omega_N \). Every block can have two types of boundaries; an external boundary, coinciding with \( \Gamma \), or an internal boundary (interface), where block \( i \) is connected to block \( j \). The latter boundaries are denoted by \( \Gamma_{ij} \). The subdomains can either have an overlap area (in which case the overlap area between block \( i \) and block \( j \) is denoted by \( \Omega_{ij} = \Omega_i \cap \Omega_j \neq \emptyset \quad \forall i,j, i \neq j \)) or only share the same boundary. Both approaches are depicted in figure 4.3.

![Diagram showing overlapping and non-overlapping decompositions](image)

(a) Overlapping decomposition of the domain into two blocks  
(b) Non-overlapping decomposition of the same global domain into three blocks

**Figure 4.1:** Different block decompositions for the same physical domain

The overlapping decomposition is called the *Schwartz method*, whereas the non-overlapping decomposition is called the *Neumann-Dirichlet method*.

The Neumann-Dirichlet method

If the interfaces of the blocks co-incide, as depicted in figure 4.3(b), an additional equation has to be solved for the unknowns on the interfaces, since these do not
belong to any specific block. It can be shown that equation 4.1 is satisfied on the whole domain if it is solved on all subdomains and $\varphi$ as well as the normal derivative across the interface $\partial \varphi/\partial n$ are continuous.

In this class of methods one usually first solves for the unknowns on the interface. The system describing the equation for these unknowns is usually called the Shur Complements system, although Chan and Goovaerts (1992) show that this matrix can also be related to Schwartz' type of methods for special cases. After having solved the interface equations all subdomains can be solved independently and concurrently. That is why nowadays much effort is put in finding improved methods for solving the Shur complements system. A good state-of-the-art introduction on theoretical considerations can be found in the proceedings of the 'Domain Decomposition' conferences, e.g. Glowinski et al. (1988), Chan et al. (1989), Keyes et al. (1992) and Mandel et al. (1997). This approach was not applied in this study, as it requires explicit treatment of the interfaces.

The Schwartz method

In this case there are no real difficulties. Since parts of the domain overlap, some parts of the domain are computed twice. Obviously this causes some computational overhead, but it increases the stability of the convergence. The basic Schwartz method to solve the global problem then consists of the following iteration process. First choose an arbitrary initial guess $\varphi^0$ in all blocks, and then construct a sequence of approximations $\varphi_1, \varphi_2, \ldots$. Each outer iteration (see section 3.3) now looks as follows:

1. Do for $i = 1, \ldots N$
2. Do for $j = 1, \ldots N$
4. Obtain boundary values on all boundaries $\Gamma_{ij} \Rightarrow$
5. $(\varphi_i^{new})_{\Gamma_{ij}} = \varphi_j^{new} \quad j < i$ (Boundaries of already computed blocks)
6. $(\varphi_i^{new})_{\Gamma_{ij}} = \varphi_j^{old} \quad j > i$ (Boundaries of not yet computed blocks)
7. End do $j$
8. Solve $\varphi_i^{new}$ on $\Omega_i$ with these boundary values
9. End do $i$
10. Check convergence; if not attained, repeat from line 1.

where $(\varphi_i^{new})_{\Gamma_{ij}}$ are the values on the interface $\Gamma_{ij}$, provided block $i$ and $j$ share an interface. Here all blocks are handled consecutively, one after the other. This means that the order in which the blocks are numbered becomes important, but the coupling between the blocks is quite good. This variant in called multiplicative Schwartz method. If we take $\varphi_j^{old}$ instead of $\varphi_j^{new}$ in step 5 we obtain the so called additive Schwartz method. In this approach we do not use the most recent values of
the boundary values, but we use the values from the previous outer iteration for all variables in all blocks. This method allows the solving of the blocks concurrently and is hence far more suited for parallelisation.

### Algebraic framework

The multiplicative and additive Schwartz method are related in the same way as the Gauss-Seidel solver is related to the Jacobi solver. This can easily be seen if we write down the algebraic, discretised version of equation 4.1, viz. equation 3.27 for the multi-block computations.

\[
A \varphi = B \Leftrightarrow \begin{bmatrix} A_{11} & \cdots & A_{1N} \\ \vdots & \ddots & \vdots \\ A_{N1} & \cdots & A_{NN} \end{bmatrix} \begin{bmatrix} \varphi_1 \\ \vdots \\ \varphi_N \end{bmatrix} = \begin{bmatrix} B_1 \\ \vdots \\ B_N \end{bmatrix} \tag{4.2}
\]

Here \( \varphi_i \) denotes all unknowns residing in block \( i \) and \( A_{ij} \) denotes the influence of block \( i \) on the unknowns in block \( j \). Hence \( A_{ii} \) denotes the discretised system for all unknowns within a single block, and the off-diagonal matrices denote the influence of unknowns in other blocks on this block. They represent the coupling of the domains. The matrices \( A_{ij} \) are zero if two blocks do not overlap, and yield the coupling of the two blocks if they overlap. The equation for the unknowns in block \( i \) can be written as

\[
\sum_{j=1}^{N} A_{ij} \varphi_j = B_i \quad i = 1, \ldots, N \tag{4.3}
\]

This system, which is equivalent to equation (4.2), has to be solved iteratively, as mentioned in the previous subsection. The general iteration process for the multiplicative Schwartz method becomes:

\[
\varphi_i^{new} = A_{ii}^{-1} \left( B_i - \sum_{j=1}^{i-1} A_{ij} \varphi_j^{new} - \sum_{j=i+1}^{N} A_{ij} \varphi_j^{old} \right) \tag{4.4}
\]

whereas the additive method is found by simply replacing \( \varphi_j^{new} \) on the right-hand side of 4.4 by \( \varphi_j^{old} \). This explains why additive Schwartz is often referred to as block Jacobi domain decomposition. If the general matrix \( A \) from 4.2 is formally written as the sum of two arbitrary matrices (\( A = S - T \)), the iteration procedure 4.4 can formally also be denoted as:

\[
S \varphi^{new} = T \varphi^{old} + B \tag{4.5}
\]

For block Jacobi \( S_J = \) diagonal part of \( A \), whereas for block Gauss-Seidel \( S_{GS} = \) lower triangular part of \( A \).
Comparison between additive and multiplicative Schwartz.

Only the additive Schwartz (or block Jacobi) method is suitable for (efficient) parallelisation and since we intend to use the domain decomposition as a means of parallelisation, we decided to use this additive Schwartz method. The difference in convergence between the multiplicative Schwartz (block Gauss-Seidel) and additive Schwartz (block Jacobi) method gives an indication of the convergence loss because of the parallelisation.

The domain decomposition will always lead to slower convergence, because of the fact that the overall system is discretised using an implicit time stepping method, which treats all values in the system as unknown and solves for them. Domain decomposition always replaces some of those unknowns by guesses (usually the values from the latest iteration) and treats those as known. The convergence deterioration because of using more blocks will be treated in section 4.5.

It can be shown (Young, 1971) that the convergence speed of an iterative method is proportional to the spectral radius of the amplification matrix $S^{-1}T$, $\rho(S^{-1}T) \equiv \max | \lambda_i |$, with $\lambda_i$ the eigenvalues of the matrix. The spectral radius of the Gauss-Seidel amplification matrix is the square of the Jacobi spectral radius, hence the Gauss-Seidel method will convergence twice as fast. We hence estimated this convergence loss of parallelisation to be a factor of two.

The multiplicative Schwartz is by definition not suitable for parallelisation. In practise we do not have any choice but to use the additive Schwartz method. Still it is interesting to see if the theoretically predicted convergence loss is observed in practise. Therefore we solved the convection-diffusion equation (3.39), explained in section 3.6. A $64^2$ uniform grid was chosen and different numbers of inner iterations in the SIP solver were applied, reaching from 1 (inaccurately) to 10 (accurately). Iterations continued until the relative error for the unknown $\varphi$ was smaller than $10^{-8}$. The relative error was defined as $||\varphi - \varphi_{\text{exact}}||/||\varphi_{\text{exact}}||$, with $\varphi_{\text{exact}}$ being the exact solution, as given by equation 3.40. The underrelaxation parameter $\alpha$ was fixed at $\alpha = 0.5$. All runs converged without any problems. Results are quoted in table 4.1.

<table>
<thead>
<tr>
<th>Type</th>
<th># Global iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SIP (1)</td>
</tr>
<tr>
<td>Single block</td>
<td>70</td>
</tr>
<tr>
<td>Two blocks, Gauss-Seidel</td>
<td>120</td>
</tr>
<tr>
<td>Two blocks, Jacobi</td>
<td>150</td>
</tr>
<tr>
<td>Jacobi/Gauss-Seidel</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Table 4.1: Difference between additive (Jacobi) and multiplicative (Gauss-Seidel) Schwartz block decomposition for different subdomain accuracies.
Obviously the Jacobi method needs more iterations to converge. However, as the accuracy of the solver becomes higher, the difference between Gauss-Seidel and Jacobi becomes larger. This is because the values on the boundaries will have changed more in every block than with less accurate solving of the blocks; hence it is more beneficial to use the latest available values. For parallelisation it might therefore be beneficial not to solve the systems in all blocks too accurately (as was also suggested by others, e.g. Brakkee (1996)), since this might lead to large changes of the values on the interfaces. For the asymptotic case, in which the subdomains are solved very accurately, the Jacobi method is approximately 2.8 times slower than the Gauss-Seidel method, which is somewhat higher than predicted. It must be said though that the way of implementing the interface condition also influences the results. This implementation will be handled in the next section.

4.4 Implementation of the domain decomposition

4.4.1 Creating the blocks

For the domain decomposition the grid-embedding technique as explained by Coelho et al. (1991) is used. This means that one global (coarse) grid is defined, and the domains are defined as subdomains of this coarse grid, with the block boundaries coinciding with the control volumes. Every subdomain can be refined with respect to the coarse grid, with the restriction that adjacent grids can be either twice as fine or coarse (in all three directions) as the grids in neighbouring blocks. This is depicted in figure 4.2.

![Figure 4.2: Domain decomposition with locally refined grid. Three levels of refinement are shown.](image)

The treatment of where to refine the grid is elucidated in section 4.7. Here we show the grid-embedding technique and show the way to build a general and flexible
domain decomposition tool.

After the definition of the block boundaries, the size of the overlap needs to be determined. We decided to use the Schwartz method with minimal overlap. This means that the blocks have an overlap of one node, resulting in a minimisation of the amount of redundant computations. In fact, all points are only computed by one specific block and the points that are located at the interface are then copied into the virtual cells (or halo cells) of adjacent blocks. (This is what makes Brakkee (1996) remark: "Neumann-Dirichlet = Schwartz, non-overlapping = minimal overlap, minimal overlap = real overlap, Schwartz = Shur".) Figure 4.3 shows the decomposition of the domain into four blocks with the halo cells. In this study one shell of

Figure 4.3: decomposition of the domain into 4 blocks with the halo-cells (shaded). The indices denote the location of the volume. The dashed lines denote the block interfaces.

halo cells is used. This means that at the block interfaces it is impossible to use second order upwind schemes, since this would require two shells of halo cells. Hence at the internal boundaries only the compact schemes (the first-order upwind, CDS
and hybrid scheme) were applied. Corner points are not used as halo cells. This would (especially in three dimensions) lead to much more communication between the blocks and communications between blocks that only share one line or even one point in three dimensions. To avoid this, the values in the corner points are obtained by extrapolation of the other virtual points and internal points.

4.4.2 Connecting the blocks

To actually connect the blocks, the halo cells should receive the value of the corresponding cells in the adjacent block. If two neighbouring blocks have the same refinement level the values on the internal boundaries are simply copied into the halo cells of the neighbouring domains. This way of coupling renders the need for explicit boundary conditions (as needed in the case of Neumann-Dirichlet) on the internal boundaries superfluous.

If the blocks have different refinement levels tri-linear interpolation is applied for all variables as shown in figure 4.4 in two dimensions for clarity. Interpolation is employed to save the band structure of the matrices. This means we can still use the standard solvers to solve the system, even if the cells adjacent to the boundaries are connected to more than 6 neighbours. The alternative option would be to create a data structure that can handle locally irregular grids and consequently a solver that can handle this irregularity. This option is more favourable as it yields a more natural coupling of the domains. However, the possibility to apply standard solvers was considered more important, hence the option mentioned above was used.

In the collocated approach all variables are stored in the centre of the control volume, hence this poses no problems whatsoever. In the staggered approach one
velocity component resides on the interface. Boerstoel (1997) demonstrates how to treat this velocity component. The fluxes and pressure gradient are stored on the interfaces. In case of different refinement levels the fluxes are computed on the fine grid side of the interfaces only, and then added and sent to the coarse block. This yields a flux conservative scheme over the block interface. These values, received from the adjacent blocks for the virtual points in the equations, can then be inserted in the equations, thereby eliminating the unknowns on the block interfaces, similar to the elimination of external boundary points.

This can be done in different ways, which can be described by looking at the difference equations themselves. The general equation for any internal point is given by equation (3.10). Consider figure 4.3: Node number (5,3) in block 1 is an internal point to block 1; we call this node P. Then node E, being the point (5,4), belongs to adjacent block number 2. Block 1 can not access its value directly, but can only use the value that block 2 has copied into the halo cell (5,4) of block 1 as an approximation to the value of \( \varphi_E \). Likewise block 1 sends the value of its point P, which is a virtual west point to block 2, to the halo cell (5,3) in block 2 to be used as the approximate value for \( \varphi_W \) there.

The virtual point E in block 1 is eliminated by simply adding the value \( a_E \varphi_E \) to the right-hand-side vector, and summing over the internal points only. In this way the internal block boundaries can be treated identical to the external block boundaries, which makes the programming transparent. This elimination is expressed as follows (the starred quantities denote the approximate values in the halo cells for the variables):

\[
a_P \varphi_P = \sum_{WNSTB} a_{nb} \varphi_{nb} + (b_P + a_E \varphi_E^*)
\]

(4.6)

We call this the straightforward update method. This manner of internal boundary implementation loosens the coupling between the different blocks and can hence cause severe convergence deterioration. Carvalho et al. (1991) also mentions this and give a more sophisticated boundary update; the term \( a_E \varphi_E \) is not added to the right-hand-side vector immediately, but the flux is added to the flux in point P:

\[
a_P \varphi_P - a_E \varphi_E^* = \sum_{WNSTB} a_{nb} \varphi_{nb} + b_P + a_E \varphi_E^* - a_E \varphi_P^*
\]

\[
\varphi_{P} \rightleftharpoons (a_P - a_E) \varphi_P = \sum_{WNSTB} a_{nb} \varphi_{nb} + b_P + a_E \varphi_E^* - a_E \varphi_P^*
\]

(4.7)

This approach establishes an implicit (strong) link through the coefficients \( a_{nb} \) rather than an explicit (weak) link through the source terms, obtained by the straightforward update. This method is called the flux modification method.

The flux modification method was compared to the straightforward boundary update method by solving the convection-diffusion equation (3.39), same grid, decomposition, solver and convergence criterion as the last section. Figure 4.5 shows
the differences between the two methods for different values of the underrelaxation parameter $\alpha$ for two blocks.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figures/figure4.5.png}
\caption{Convection-diffusion equation on $64 \times 64$ nodes using 2 blocks and the SIP-solver. Influence of the under-relaxation on the convergence behaviour. (a) Straightforward update (present method) (b) Flux modification model (Carvalho, 1991).}
\end{figure}

The straightforward method is a very robust and stable update. More accurate solving of the subdomains leads to faster global convergence. This effect is mainly due to the weak coupling of the unknowns over the interfaces. In this way the weak coupling works as an additional underrelaxation.

The flux modification method leads to a stronger coupling of the blocks, resulting in a faster convergence than the straightforward flux method, if the method converges. It performs worse for more accurate solving of the subdomains. This can be explained by stating that the flux modification method was not designed to deal with accurately solved subdomains, but rather was developed for accurate connecting two blocks with different refinement level. Carvalho \textit{et al.} (1991) only performed one inner iteration. It can be seen from figure 4.5 that when solving the subdomain by performing a single iteration, the method is very fast. It converges in approximately 70 steps, whereas the straightforward update requires 150 steps. However, if the domains are solved concurrently this method introduces an approximation error, which makes the method quickly unstable if the domains are solved more accurately, unless we apply more severe underrelaxation. The method is not particularly suited for fast and accurate solving of the subdomains. Hence it was decided to retain the straightforward update.
4.4.3 Computing the blocks

To complete the extension to a multi-block code a so-called topology file is added. This file is generated during the construction of the grid and the creation of the blocks. It contains all the information about the topology of the domain decomposition, such as the way in which the blocks are connected, with which sides they connect, and which indices of gridcells are involved. Every block knows the contents of this topology table and can hence decide which points to update from which block.

![Diagram of the SIMPLE iteration process]

**Figure 4.6:** Flowchart for the FURNACE-program.

The last step in building a domain decomposition code is to extend the single-block code used in previous studies (see chapter 1) to a multi-block code by implementing the internal boundary conditions as described in the last subsection. Apart from this the existing single-block code only needed to be adapted slightly to incorporate con-
vergence checking. The flowchart for a particular block in the multi-block algorithm is given in figure 4.6. The circles marked GC and LC denote respectively Global Communication (i.e. this block needs information from all other blocks) and Local Communication (i.e. this block only needs information from its adjacent blocks). Note the resemblance with figure 3.4. The terms inner- and outer iterations, as well as the SIMPLE iteration process have been discussed in section 3.3. The remaining loop, the Nsolve iteration loop will be explained in the next section.

Once the topology file is created and a communication protocol has been set up, this domain decomposition approach remains extremely flexible. The blocks can be coupled in a completely arbitrary manner, the local grid refinement is handled in the communication layer. In principle different models can be used within different blocks. In this way also the extension to parallel computing, as will be discussed in chapter 5, becomes just a matter of slightly modifying the communication protocol to enable the use of running parts of the code in parallel and on different machines. If all blocks are solved on one processor communication is just a data copy. In this case an additional loop over the blocks is added, but this has not been depicted in the flowchart, since it would not alter the main feature of the multi-block code.

4.5 Convergence behaviour of the multi-block algorithm

4.5.1 The reason for convergence loss

Since the convection-diffusion equations are all elliptic in space, the value of a particular quantity in a particular location in the domain depends on the values in the entire domain. In discretising the equations this ellipticity is approximated by taking into account the values in the neighbouring cells only. The differential operator is replaced by a difference stencil, the size of which determines which cells are taken into account in the computation of the new value.

If the solution to the difference equation were obtained using a direct method on a single-block grid, this locality of the difference operator would not lead to any problems. However, in a multi-block computation the difference stencil addresses quantities in another block. This is shown in figure 4.7, where point E is located in block 2. The point E resides in the halo-cell of block 1. Hence the value in this node can only be used, and not updated or modified. A solver that relies on the 'right' to be able to modify the neighbour values will consequently always suffer from convergence loss. If we make sure that the difference stencil never extends into the other block, or the values in the other block are only needed explicitly, then indeed no convergence loss between a single-block and a multi-block solution will be found. Explicit time stepping and explicit solvers (like point Jacobi) have this feature. The disadvantage of these solvers is that the convergence characteristics of these solvers are usually quite
bad. For explicit time integration the time step should be small enough to ensure a numerically stable algorithm. In turbulent combustion simulations, more advanced solvers are mandatory and implicit time integration has to be used. The stencils for these solvers extend into the other blocks and hence convergence loss will occur.

Most Krylov subspace methods (and the SIP-solver) have the feature that the values are only updated at the end of every iteration cycle. In principle the use of these solvers can yield a domain-independent code. However, to achieve this the solvers have to be extended to take into account the local grid refinement in the other block, and much communication has to be done in the solver, which may significantly increase the time per iteration. The total efficiency of an algorithm is the product of the amount of iterations times the time per iteration. At the start of this study, when only workstations were available, communication was very expensive. Hence this option has not been applied in the present study. However, when more advanced machines became available, we implemented an improved version of the solver, which is discussed in section 4.6, which still saves the structure of the matrix.

For an iterative solver to yield the proper answer, information needs to travel back an forth through the global domain several times to take into account all phenomena in the domain. Unless the unknowns on the interface are updated immediately after being computed, this traveling of information through the global domain will be prohibited by the domain decomposition. As a coarse estimate we state that the convergence deterioration will be linearly proportional to the number of blocks.

The choice for an updating mechanism of the unknowns on the block interfaces influences the convergence loss. The paragraph about the comparison between the additive and multiplicative Schwartz method already showed that the choice of any multi-block iteration method is a cause of the convergence loss. In the previous section

Figure 4.7: Location of nodes near a block interface.
we saw the difference between the treatment of the unknowns on the interface. It is possible to attain a better coupling mechanism by actually solving for the unknowns on the interface rather than just updating them from neighbouring blocks. This is done by formulating a so called influence matrix which yields the system of equations for unknowns on the interface. Brakkee (1996) explains this method. We did not use this technique because it turned out that the Schwartz updating method is usually much more robust and equally good.

**Other convergence-behaviour determinating factors**

The behaviour of the solver in the blocks will differ because in fixed-size problems the number of CVs in the blocks will become smaller as the number of domains increases. This will improve the performance of the solver. However, in section 4.4.2 it was shown that better performance of the solver in the blocks might not lead to overall better performance due to large differences for the unknowns on the block interfaces between updates. For scaled-size problems usually the number of CVs in a blocks stays constant when more blocks are added, and this solver improvement is not observed.

Several techniques can be applied to restore the coupling between the domains. Multigrid acceleration and Krylov subspace acceleration are the most well-known ones. The multi-block multigrid technique seems to be the most successful in complex geometries. It has been applied by *e.g.* Rossow (1992), Perng and Street (1991), Baysal *et al.* (1991), but all on flows without combustion. Results for complete furnace simulations have been published by Twerda *et al.* (1998). The second method, Krylov subspace acceleration, uses the domain decomposition as a preconditioner for the Krylov subspace method. This kind of accelerated domain decomposition method is usually referred to as a Krylov-Schwarz method (Keyes, 1995). Any kind of Krylov-subspace method can be used. Brakkee (1996) shows accelerations using the GMRES and GCR method. Also the settings of underrelaxation parameters might be different for the multi-block computations. However, we have found no need to change underrelaxation parameters in any multi-block computation for the solvers. No effort was undertaken in investigating the optimal underrelaxation factors as a function of the block number.

**A measure for convergence deterioration**

A quantitative measure for expressing the convergence deterioration of a multi-block code is given by Schreck and Perić (1993). Schreck and Perić (1993) came up with the term *numerical efficiency* to denote the relative increase in outer iterations due to the multi-block calculations. They couple this immediately to the parallelisation, as we will do in the next chapter. In Verweij *et al.* (1997) these terms are also used to evaluate different parts of the parallel multi-block algorithm. We will take this definition of numerical efficiency: the ratio of the number of single block outer
iterations and a multi-block computation on \( n \) blocks. The definition of numerical efficiency we will use is:

\[
E_{n}^{\text{num}} = \frac{i_{1}k_{1}}{i_{n}k_{n}}
\]

(4.8)

where \( i_{n} \) is the mean number of floating point operations per outer iteration for \( n \) blocks and \( k_{n} \) is the number of outer iterations required to reach some residual level. We assume that \( i_{n} \) is independent of the number of blocks used. When using domain decomposition in conjunction with parallelisation this assumption is a slight simplification as it ignores the fact that reading of the topology table and performing global communication is more expensive when using more blocks. However, in general this additional overhead is small.

4.5.2 Influence of the number of blocks and topology

Convection-diffusion equation

In the previous subsection a linear dependency between the number of blocks and the global convergence behaviour was suggested. The easiest test case to assert this assumption is the convection-diffusion equation (3.39). Two tests were performed: solving the subdomains accurately (SIP(10)) and inaccurately (SIP(1)). The global domain consisted of \( 64 \times 64 \) CVs. The number of domains was varied from 1 to 5. To achieve this the global domain was split in the \( x \)-direction from 1 to 5 domains. The results are depicted in table 4.2.

<table>
<thead>
<tr>
<th>#Domains</th>
<th>Domain topology</th>
<th># iter. with SIP(1)</th>
<th>with SIP(10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( 64 \times 64 )</td>
<td>70 (1.000)</td>
<td>6 (1.000)</td>
</tr>
<tr>
<td>2</td>
<td>( 2 \times 32 \times 64 )</td>
<td>150 (0.467)</td>
<td>72 (0.083)</td>
</tr>
<tr>
<td>3</td>
<td>( 2 \times 21 \times 64 + 1 \times 22 \times 64 )</td>
<td>227 (0.308)</td>
<td>113 (0.053)</td>
</tr>
<tr>
<td>4</td>
<td>( 4 \times 16 \times 64 )</td>
<td>303 (0.231)</td>
<td>145 (0.041)</td>
</tr>
<tr>
<td>5</td>
<td>( 4 \times 13 \times 64 + 1 \times 12 \times 64 )</td>
<td>353 (0.198)</td>
<td>173 (0.035)</td>
</tr>
</tbody>
</table>

Table 4.2: Numerical efficiency for the convection-diffusion equation on a global \( 64 \times 64 \) grid using accurate and inaccurate solving. The global domain was split in the \( x \) direction from 1 to 5 domains. The value in parenthesis is the numerical efficiency.

Note that we have specifically split the blocks in the \( x \)-direction only. In the next testcase different topologies will be addressed. This testcase shows that, using the inaccurate solving (SIP(1)), the amount of outer iterations indeed increases proportional to the number of blocks used. For the accurate solving (SIP(10)) however, the single-block convergence behaviour is so good (only 6 iterations are needed for
this case) that the numerical efficiency drops immediately when using more blocks (72 iterations are needed when using two blocks already). This stems from the fact that the conservative boundary update method is used, which was already shown to slow down the convergence. Another contribution to this behaviour is the fact that with accurate solving, the system keeps on computing using old interface values much longer, deteriorating the solution rather than improving it.

Laminar flow

The laminar flow in the lid-driven cavity by Shih and Tan (1989) was taken as the most elaborate testcase to see the influence of the number of blocks and the topology on the global convergence behaviour. Uniform meshes of size $16 \times 16$, $32 \times 32$ and $64 \times 64$ were adopted, using the SIP(1) solver. Each mesh was decomposed in a number of blocks and a number of topologies. 1 to 16 blocks were chosen, which were decomposed into strips (i.e. domain decomposition in one direction) and blocks. Iterations continued until the change in the relative difference with respect to the

![Graph showing dependency of the convergence behaviour on the number of blocks and the topology of the domain decomposition for the laminar cavity flow of Shih (1989). The SIP(1) solver was used. Shown are results for a $16 \times 16$, $32 \times 32$ and $64 \times 64$ uniform mesh.](image)

**Figure 4.8:** Dependency of the convergence behaviour on the number of blocks and the topology of the domain decomposition for the laminar cavity flow of Shih (1989). The SIP(1) solver was used. Shown are results for a $16 \times 16$, $32 \times 32$ and $64 \times 64$ uniform mesh.
Figure 4.8: (cont.) Results for $32 \times 32$ and $64 \times 64$ uniform mesh.
Figure 4.9: Dependency of the convergence behaviour on the number of blocks and the topology of the domain decomposition for the laminar cavity flow of Shih (1989). The SIP(1) solver was used. Shown is the numerical efficiency for a 16 x 16, 32 x 32 and 64 x 64 uniform mesh.

\[
\sum_{i,j=1}^{N_x,N_y} \frac{\|\varphi(i,j) - \varphi_{exact}(i,j)\|}{\|\varphi_{exact}(i,j)\|}
\]

(4.9)

was less than $10^{-6}$. The amount of iterations needed to reach the converged solution is plotted in figure 4.8 for the 3 different gridsizes. In this plot the points that are connected belong to the same number of domains, but each with different topology.

All grids show an increase in outer iterations for increasing number of blocks. However, the increase is not as severe as observed for the convection-diffusion equation. This is caused by the fact that for the convection-diffusion equation the strip-wise decomposition was chosen. From all the testcases it becomes clear that the decomposition of the domain into strips always results in more unstable convergence behaviour. The number of blocks should be as uniform as possible in all directions for minimal
influence. Figure 4.9 shows the numerical efficiency for the computations discussed above. The numerical efficiency is almost independent from the number of nodes in the global domain. Apart from the pathetic cases (the 16×16 domain split in 16×1 block will have only one CV in each block) the numerical efficiency is more influenced by the topology of the domain decomposition than by the number of CVs in the domain. This is a very lucky feature of the model, since it suggests that the convergence deterioration because of adding more blocks is not as bad as predicted by the convection-diffusion equation. The average numerical efficiency does decrease with the number of blocks, but the dependency is definitely not linear. From figure 4.9 it can be seen that the efficiency drops down to 92% for 16 (4×4) blocks.

Much more important is the observation that the convergence behaviour decreases if the number of blocks in one direction differs significantly from the number of blocks in another direction. This effect can be explained using the results of the last section. The more blocks there are in one direction, the more iterations are needed for the information to travel through all the blocks, hence the slower the convergence. Brakkee (1996) also finds the decomposition of strips less beneficial than the decomposition into blocks. In section 4.6 we will address ways to improve this convergence behaviour.

Turbulent reacting flow

Next, the block decomposition is applied to the entire furnace simulation. In this case it is not possible to start from scratch, since this would lead to enormous density-gradients, and will never lead to a convergent solution. A proper starting field is absolutely necessary for these computations. Hence it was decided to first create a decent starting field, with all quantities (especially density) already having a reasonable value, before the convergence tests were applied. This starting solution was then used for all computations by interpolating this solution onto all other block decompositions. For this test the 'medium grid' of 32 × 50 × 50 control volumes was used. Table 4.3 lists the decompositions used for the computations. Note again that the

<table>
<thead>
<tr>
<th># Domains</th>
<th>Decomposition (Nblx × Nbly × Nblz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1×1×1</td>
</tr>
<tr>
<td>2</td>
<td>1×2×1</td>
</tr>
<tr>
<td>4</td>
<td>1×2×2</td>
</tr>
<tr>
<td>8</td>
<td>2×2×2</td>
</tr>
<tr>
<td>16</td>
<td>2×4×2</td>
</tr>
<tr>
<td>32</td>
<td>2×4×4</td>
</tr>
</tbody>
</table>

Table 4.3: Different block decompositions used in the IFRF furnace simulation. A global 32 × 50 × 50 grid was used with the SIP(1) solver.
4.5. **Convergence behaviour of the multi-block algorithm**

$x$-direction specifies the width of the furnace, $y$ specifies the length and $z$ the height. For clarity figure 4.10 sketches the resulting decomposition into 16 blocks. For this decomposition into 16 blocks the blocks 1 to 8 lie near the symmetry plane, the uneven blocks lie near the glass bath and the even blocks occupy the upper half of the combustion chamber. The gas-inlet is situated in block 1, the air inlet in block 2 and the outlet in block 8. We have applied a decomposition which is as uniform as possible, as tests above have shown that this leads to the highest numerical efficiency. Iterations were continued until the mass- and fuel balances, as defined in section 3.3.3 had dropped below 1%. The SIP(1) solver was used. The results are shown in figure 4.11 for 1, 2, 4, 8, 16 and 32 blocks. These results indicate that also for the turbulent combustion case the influence of the number of blocks on the convergence behaviour is modest. The scaling of the figure is somewhat misleading. It must be said that proper comparison is slightly unfair because the interpolation from one decomposition to the other causes some small perturbations in the solutions. However, we can conclude that the influence of domain decomposition is limited to no more than 20% for computations up to 32 blocks.

**The quest for an optimal decomposition**

The results in the last section show that the multi-block convergence behaviour depends on the case studied. For industrial applications like the furnace simulations, the system of equations is quite stiff and different interacting models determine the numerical behaviour. Little influence of the direction of the mean flow on the conver-
(a) Influence of the number of blocks on the convergence behaviour

(b) Numerical efficiency

Figure 4.11: Influence of the number of blocks on the convergence behaviour for the IFRF furnace simulation. A global $32 \times 50 \times 50$ grid was used with the SIP(1) solver. (a) Number of iterations. (b) Numerical efficiency.
gence behaviour was observed. However, during single-block turbulent combustion simulations it was found that some variables converged slower than other variables. In particular, the convergence behaviour for the $u$-velocity, which is the velocity component normal to the sides of the furnace, was usually much slower than for other velocity-components. This was caused by the symmetry plane at $x = 0$.

This raises an interesting question. Perhaps these variable are also more sensitive to the block decomposition than others. In particular, the convergence of the $u$-velocity might be slower when more blocks were used in the $x$-direction, and similar for other velocity components.

To see the effect of block decomposition on different variables we focused on the full furnace simulation, with the same 16-block decomposition as above. We investigated the residuals of the three velocity components and the pressure. Figure 4.12 shows the scaled residuals, according to criterion 3.22. 5000 iterations were performed.

![Figure 4.12: Residuals for velocity and pressure in the different blocks for the furnace simulation. 5000 iteration were performed.](image)

The block decomposition influences some variables in a different manner than others, indeed as expected. The residual in the $u$-velocity decreases fastest in the blocks near the symmetry plane, whereas the $v$-velocity (along the furnace) is best predicted in the odd blocks. This means that the decomposition beneficial for one
variable will decrease the convergence speed for another variable. There is no overall optimal decomposition possible. This is probably also the explanation why the decomposition into 'blocks' rather than strips works the best for most cases. The pressure seems to be less influenced by the block decomposition because the elliptic nature of pressure makes it in principle not suited for domain decomposition, unless a suitable acceleration technique is applied.

4.6 Improvement of the multi-block convergence behaviour

The frequency with which the values of the unknowns in the halo cells are updated largely determines the influence of the domain decomposition on the numerical stability. Frequent updating of the halo cells will lead to a better coupling between the blocks. So far the values in the halo cells have only been updated after every outer iteration. This means that during an entire outer iteration the values from the previous iteration in another block are used, whereas they have been updated in the block in which they are computed. For improvement of the multi-block convergence behaviour it might be beneficial to update the variables in the multi-block simulation every time they have been recomputed in the other block. This will of course lead to much more communication overhead in case of parallel execution, as will be discussed in the next chapter, but this will lead to a more stable numerical algorithm. Furthermore we will see that on parallel machines with a fast network, the communication overhead is negligible with respect to the load imbalance and the computation time. The only way to get a completely block-independent convergence behaviour is by either applying point-Jacobi solvers or by using Krylov subspace methods (or similar) which only use the values in the neighbouring point and do not update them. However, as mentioned, this would require severe code modification and requires significantly more communication than the method suggested here.

The obvious place to update the values in the halo cells is after the solving of the system. If we take this approach one step further, we could expect that if we would immediately resolve the variable with these newly updated values on the boundary, the process might be even more stable. Hence we added an iteration loop over the solving of all variables. This is called the $N_{solve}$ iteration loop and is depicted in figure 4.6. This approach is discussed in section 4.6.1.

The last place where the hydrodynamical variables are updated is after the velocity-corrections in the SIMPLE-algorithm. Updating the velocities and pressure after their corrections would be an interesting idea too. Also here the opportunity for repeating the SIMPLE iteration comes into play. This iteration cycle is called the $N_{simple}$ iteration cycle and is depicted too in figure 4.6. As the result of the $N_{solve}$ variable was already very satisfactory, no attempt was made to improve the convergence behaviour via changing the $N_{simple}$ parameter. The pseudo code for the new iteration process
in every block now becomes as follows.

Do until converged
  Do Nsimple times    # The Nsimple iteration process
    Do nsolve(U) times
      solve (U-velocity)
      Update halo-cells for U-velocity
    Enddo
  Do nsolve(V) times
    solve (V-velocity)
    Update halo-cells for V-velocity
  Enddo
  Do nsolve(W) times
    solve (W-velocity)
    Update halo-cells for W-velocity
  Enddo
  Do nsolve(P) times
    solve (Pressure)
    Update halo-cells for pressure
  Enddo
  Update velocities and pressure
  Update halo-cells for velocities and pressure
 Enddo
 Do for all independent variables
  Do nsolve( independent variable) times
    solve ( independent variable)
    Update halo-cells for independent variable
  Enddo
 Enddo
 Compute residuals
 Enddo

4.6.1 Variable-iterations increase

Apart from increasing the number of inner iterations, increasing the amount of times a variable is solved yields a handy tool to finetune the iteration process. For some variables, like mixture fraction or pressure, an accurate solution is more desirable in every outer iteration than for others. The idea is that multi-block iterations will only transfer the solution one block per iteration, so that every variable should at least be solved as often as there are blocks in one direction to make sure that information has traversed all the blocks. Of course more often solving will increase the coupling between the blocks even more.
First the difference between increasing the \textit{Nsolve} parameter and increasing the number of iterations should be mentioned. Both seem to do the same thing. The difference is, however, that increasing the number of inner iterations means a more accurate solving of the system $A\varphi = b$ (viz. equation 3.21). Increasing the \textit{Nsolve} parameter means that the values on the interfaces are updated and the system $A$ with right-hand-side $b$ is build again. Of course this takes significantly more time (building of matrix can be as time consuming as solving of matrix if only one or two iterations in the solver are done). The wall-clock time is an important factor in deciding whether the increase of the \textit{Nsolve} parameter indeed leads to quicker (in the sense of CPU-time) convergence. However, in turbulent combustion simulations the stability of the method is very important too, so even if a slight increase in CPU-time is found, this would not be a reason to abandon this approach.

![Graph showing number of iterations](image.png)

\textbf{Figure 4.13}: Number of iterations for convergence of lid-driven cavity simulation proposed by Shih (1989). A uniform $16 \times 16$ grid was used with the SIP solver. The solid line shows the effect of increasing the \textit{Nsolve} parameter and setting the amount of inner iterations to 1. The dotted lines shows the effect of increasing the amount of inner iterations and setting the \textit{Nsolve} parameter to 1.
The laminar lid-driven cavity flow with the exact solution as proposed by Shih was used again as a first testcase. Figure 4.13 shows the influence of increasing the \textit{Nsolve} parameter and increasing the number of inner iterations for a single-block domain. This will hence give a good indication of the difference between computing longer with the same matrix or updating the matrix after one inner iteration. This figure shows that increasing the amount of inner iterations really doesn't contribute to a faster global convergence. This has been mentioned in the previous chapter too, and can be explained by noting that the system being solved is only an approximation to the difference equation that has to be solved. When increasing the amount of inner iterations, eventually the (approximate) system is solved exactly, but no gain in convergence speed is noticed. On the other hand, when updating the right hand side and rebuilding the matrix again, even very poor solving (1 inner iteration) leads to improved convergence behaviour when resolving the system. It can be concluded that, from a stability point of view it is much more attractive to have a poor solver and update the matrix as often as possible than driving the inner iterations to a very tight convergence. For the multi-block convergence behaviour is was already shown

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4_14}
\caption{Dependency of the convergence behaviour on the number of blocks and the topology of the domain decomposition for the laminar cavity flow of Shih (1989). The SIP(1) solver was used. Shown is the influence of increasing the Nsolve parameter for a $64 \times 64$ uniform mesh.}
\end{figure}
that solving the system accurately is more unstable, and also the coupling between the blocks will not be improved by increasing this parameter. Hence it was decided to perform all subsequent tests with 1 inner iteration for the solver and 2 to 5 for the pressure-correction. The slightly higher amount of iterations for the pressure can be explained by the fact that the first guess is always zero. This, combined with the fact that the pressure equation is very elliptic, makes somewhat more accurate solving beneficial. Figure 4.14 shows the impact of increasing the Nsolve parameter. The effect of rebuilding the system and updating the halo cells is apparent. The more often this is done, the more stable the iteration process becomes. Whereas increasing the amount of inner iterations quickly becomes ineffective and does not change the overall convergence behaviour of the code too much, increasing the Nsolve parameter does stabilise the system very effectively. There does not seem to be an apparent bound on the number of iterations from an efficiency point of view.

![Graph showing the effect of Nsolve parameter]

**Figure 4.15:** Effect of increasing the Nsolve parameter for the furnace simulation. A global $32 \times 50 \times 50$ grid was used with the SIP(1) solver, divided in $8 \ (2 \times 2 \times 2)$ and $16 \ (2 \times 4 \times 2)$ blocks.

Next, the effect of increasing the Nsolve parameter on the convergence behaviour of the furnace simulations was investigated. We have already seen that most solvers do perform well on the straightforward cases with simple grids, but have rather poor
behaviour on complex flows with highly non-uniform grids. Hence it is interesting to see if the effect of increasing the amount of $Nsolve$ iterations does lead to a similar improvement. Figure 4.15 shows the result for the furnace computations on 8 and 16 blocks respectively for $Nsolve$ values of 1, 2, 4 and 10. It can be seen that indeed also for this case the improvement is similar to the easier cases. The decrease in number of iterations is significant. Again there does not seem to be a lower bound to the improvement, which is the case when doing more iterations in the solver without updating the boundaries and rebuilding the system.

Finally, figure 4.16 shows the effect of increasing the $Nsolve$ parameter for the lid-driven cavity flow and the turbulent combustion case. The $y$-axis gives the relative gain in number of outer iterations. We see that for $Nsolve=2$ we need approximately $1/1.7$ times the number of iterations as with $Nsolve=1$. It becomes clear that the influence of increasing this parameter is independent of these two cases, although they are two completely different cases. Only the time to rebuild the system and update the boundaries might become prohibitive for larger values of $Nsolve$. The

![Figure 4.16: Effect of increasing the $Nsolve$ parameter for several cases; the lid-driven cavity flow proposed by Shih (1989) and the turbulent combustion case.](image-url)
absolute gain, in terms of CPU time, is given by the number of iterations times the
time per iteration, which will increase. If the time needed to rebuild the matrix
becomes too large, the relative gain of less iterations becomes smaller. A detailed
analysis of the performance of different code parts will be given in section 5.6.2.

4.7 Local grid refinement criteria

In section 4.2 two important reasons for using domain decomposition were mentioned:
Parallelisation and local grid refinement. So far we have given the framework for
the implementation of the domain decomposition, which can now be extended to
the implementation on parallel platforms. Chapter 5 will discuss this topic. This
section discusses the possibilities to use the block decomposition for obtaining more
accurate solutions in an efficient way. We have already showed in section 4.4.2 that
the implementation of local grid refinement on a block-by-block basis in a general
domain decomposition based FVM code can be done quite easily.

4.7.1 The need for locally refined grids

In a furnace, the relevant physical phenomena (like turbulence and chemistry) occur in
a broad range of lengthscales. These phenomena have to be captured by the numerical
model, and hence are directly linked to the number and distribution of grid cells over
the domain. In general this means that the predicted values suffer from truncation
errors, inducing an error $e$ between the solution of a variable onto an infinitely fine
grid ($\varphi_0$) and the numerically predicted value on a given grid ($\varphi_i$); the error is defined
as $e \equiv \varphi_0 - \varphi_i$. Before we look at refinement criteria, it is interesting to look at the
quantitative influence of the grid size on the numerical results. Here two cases are
presented: The laminar lid driven cavity and the turbulent combustion case. Figure
4.17 shows the minimum value of the streamfunction as a function of the grid size
for the laminar lid-driven cavity the flow by Shih and Tan (1989). We started with a
uniform $4^2$ grid, while the finest grid consisted of $256^3$ nodes.

The relative error in the streamfunction for the $32^2$ grid is still in the order of 1 %,
and only at a uniform grid of $128^2$ the error drops below 0.05%. A better distribution
of grid points might lead to an improved behaviour, but the idea is that the user
should not give the final distribution of grid points.

For the furnace calculations, Boerstoel (1997) has done some accuracy studies.
He used four grid sizes, ranging from a 'coarse' ($16 \times 24 \times 20$) grid to a 'very fine'
($32 \times 72 \times 60$) grid. The relative change of some relevant variables (oxygen- and NO
concentration in the flue gases, the mean temperature in the outlet and the heat flux
to the glass bath) that he found are quoted in table 4.4: Although some quantities
are almost indifferent to grid refinement, especially the NO predictions are extremely
sensitive to grid refinement. For a discussion on the underlying phenomena that cause
this sensitivity we refer to Boerstoel (1997). In both these cases many gridpoints
4.7. Local grid refinement criteria

Figure 4.17: Dependency of the minimal value for the streamfunction on the gridsize for the laminar lid-driven cavity flow by Shih (1989)

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Gridsize</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$16 \times 24 \times 20$</td>
</tr>
<tr>
<td>$O_{2,\text{flue}}$</td>
<td>+17 %</td>
</tr>
<tr>
<td>$q_{\text{load}}$</td>
<td>+2 %</td>
</tr>
<tr>
<td>$T_{\text{flue}}$</td>
<td>+2 %</td>
</tr>
<tr>
<td>$NO_{\text{flue}}$</td>
<td>-33 %</td>
</tr>
</tbody>
</table>

Table 4.4: Dependency of some relevant quantities on the gridsize for the IFRF furnace simulations.

were required to obtain grid independent results. These results show that local grid refinement might offer a much better distribution of grid points and consequently lead to a higher numerical accuracy with a moderate number of nodes.

4.7.2 Possible routes to higher accuracy

The numerical accuracy can be improved in several ways; one can alter the number of grid nodes, either by increasing the total number of grid nodes (the so-called h-method), or by altering their distribution (the r-method); the latter will increase the
grid-node density in some 'interesting areas' of the flow. The \( r \)-method has not been considered, since there is no way of being sure that a given number of grid nodes is sufficient for a specific problem. Furthermore, this re-ordering of nodes can lead to highly disordered grids. Therefore the adding of cells to refine the grid has been considered here. Another option would be to increase the accuracy of the differencing scheme, but this option requires that the higher-order derivatives of the variables are sufficiently smooth. In general this cannot be guaranteed for most quantities, especially in case of turbulent combustion.

In unstructured grids, using FEM or FVM, the refinement on cell basis has proved very successful and is widely used. There is a plethora of literature on the topic. Interested readers can browse Musaferija and Gosman (1997a) or Pelletier et al. (1997). However, in structured grids, where refinement cannot easily be done on cell basis since this would ruin the structure of the matrix, local refinement is much less common still. Recently, some authors have applied locally refined irregular grid portions as block-structured patches within coarse-grid blocks (Chen et al., 1997), but this elegant and promising method is not easily implemented in an existing block-structured code. As an alternative, block-structured refinement (the refinement of a block as a whole) is applied. This approach was used by Thompson and Ferziger (1989) and is also discussed in Verweij et al. (1998a) and Verweij et al. (1998b).

Before explaining the criteria, we need to distinguish between non-uniform grids and locally refined grids. Both are shown in figure 4.18. When the amount of gridlines

![Figure 4.18: Different forms of local refinement. Left: Non-uniform grid Right: Locally refined grid](image)

on one border of the domain equals the number of gridlines at the opposite border of the domain the grid is called non-uniform. In this case some refinement can be attained by varying the space between consecutive grid lines; especially if the grid is allowed to be non-orthogonal this can lead to a very sensible distribution of grid points. However, in our study the grid is Cartesian, hence this approach will automatically
lead to unwanted clustering in other parts of the domain. This clustering is clearly visible in the left part of figure 4.18 in the oval region. The right figure indicates our definition of locally refined grids is shown. It is clear that this is a much more effective way of grid refinement. The grid is only refined where this is actually considered necessary.

4.7.3 Refinement criteria

The most important question, however, which forms the core of this section, is the question of where to refine the grid. Some error estimator should provide information about the regions where the grid should be refined, but this is in practise easier said than done. The non-linear system of coupled equations (apart from the models and their induced modelling errors) makes it complicated to state which refinement will decrease the truncation errors in all the fields. The two choices to be made in refinement are the criterion to be applied and the quantity which is used to apply the criterion on. Error estimators like Richardson extrapolation techniques are commonly used to determine refinement regions, but in most practical applications this technique is too memory demanding. We suggest other criteria, namely the gradient or curvature of some interesting quantity. These criteria can be very easily implemented in FVM-based codes and are much cheaper than classical error estimators.

a. Richardson extrapolation

Suppose \( \varphi_s \) is the exact solution for a set of equations and \( \varphi_i \) is the approximate numerical solution on the grid which reference cellsize \( h_i \). If the accuracy of the method is of order \( p \) then the following relation for the truncation error holds:

\[
\varphi_s - \varphi_i \equiv C_i h_i^p
\]  

(4.10)

Now suppose we have two grids \( h_1 \) and \( h_2 \), with \( h_2 \) the finer grid, \( h_2 < h_1 \). To estimate the error on the fine grid we can use 4.10 to write:

\[
\varphi_s - \varphi_2 = C h_2^p \Leftrightarrow \\
(\varphi_s - \varphi_2) (\varphi_2 - \varphi_1) = C_2 h_2^p (\varphi_2 - \varphi_1) \Leftrightarrow \\
(\varphi_s - \varphi_2) (\varphi_2 - C h_2^p - (\varphi_s - C h_1^p)) = C h_2^p (\varphi_2 - \varphi_1) \Leftrightarrow \\
\varphi_s - \varphi_2 = \frac{\varphi_2 - \varphi_1}{C h_1^p - C h_2^p} \Leftrightarrow \\
\varphi_s - \varphi_2 = \frac{h_1}{h_2} \left( \frac{h_1}{h_2} \right)^p - 1
\]  

(4.11)
Consequently, if grid $h_2$ is twice as fine as grid $h_1$ ($h_1 = 2h_2$), as we use in local grid refinement, then the error in node $P$, $e_P$ is given by:

$$e_P \equiv \frac{\varphi^h_P - \varphi^{2h}_P}{2^p - 1}$$  \hspace{1cm} (4.12)

The method is often used in literature and very suitable for error estimation, since it immediately links the truncation error on a coarse grid to the truncation error on a fine grid, independent of the models used in the computations. However, to be reliable, Richardson extrapolation requires a smooth solution, which means that the solution on the coarse mesh should already be sufficiently smooth to obtain reliable error estimates. For combustion problems, where geometry and physics are complex, the coarsest reliable mesh has to capture the thin reaction zone and therefore contains so many grid points that a uniform refinement, just to estimate the error, cannot be afforded. Hence this criterion has not been taken into account in the present study. However, Thompson and Ferziger (1989) use this approach on relatively simple flows and show details of the computational effort involved.

b. Truncation error estimation via higher order schemes

The error in node $P$, $e_P$ is given by:

$$\tau_P \equiv \sum_j \left[ (\rho u_j \varphi - \Gamma \varphi \frac{\partial \varphi}{\partial x_j})^H - (\rho u_j \varphi - \Gamma \varphi \frac{\partial \varphi}{\partial x_j})^L \right]$$

Solve $e_p$ from system $\mathcal{A}e = \tau$  \hspace{1cm} (4.13)

$\Gamma$ is the effective diffusivity, $\rho$ is the density and $u_i$ is the velocity field. The superscripts $H$ and $L$ denote a fourth- and second order approximation to the flux, respectively, and $\mathcal{A}$ denotes the system to be solved to obtain the quantity $\varphi$. Muzafarijia and Gosman (1997a) realised the disadvantages of the Richardson extrapolation and developed a method, similar to Richardson extrapolation, which is based on the difference of higher- and lower order approximation of the fluxes to approximate the truncation error. This method is the most expensive of the four methods mentioned, since an entire system needs to be solved for the error. Furthermore this method can only be used for independent quantities which are solved for, since a system matrix $\mathcal{A}$ has to be available for the quantity used for refinement. Hence some of the most important quantities in turbulent combustion, like density and temperature, which are not solved but derived from the mixture fraction, its variance and the enthalpy, cannot be used as criterion. The method has been implemented, but we only tested it on a laminar flow, to compare with other methods.

c. Gradient

The error in node $P$, $e_P$ is given by:

$$e_P \equiv \| \nabla \varphi_P \| \times h$$  \hspace{1cm} (4.14)
This criterion is based on the gradient of the solution. This criterion, investigated by Chen et al. (1997), reflects the belief that in complex flows steep gradients of relevant quantities also mark regions which should be refined. This criterion can very easily be implemented in the code (in fact, this operation is one of the basic operations in computation fluid dynamics), and is very cheap to perform. We have tested the performance of this criterion on different flows.

d. Curvature (Present method)

The error in node $P$, $e_P$ is given by:

$$e_P \equiv |\nabla^2 \varphi_P| \times h^2$$  \hspace{1cm} (4.15)

We suggested to try this criterion for reason of simplicity. The implementation effort is almost negligible and the computational effort, although being twice as high as the gradient criterion, is still very small. The reason we wanted to try this next to the first derivative is that the stress tensor (viz. 2.3), which contains the gradients of the mean velocities, has to be differentiated to obtain the form that appears in the momentum equation 2.2. Hence the curvature (the second derivative), which yields the areas of high shear stress gradients, might be an interesting quantity too. Apart from this, the truncation error in second order schemes is proportional to the curvature. These reasons led us to the belief that the local curvature might be a better error predictor than the local gradient.

Possible choices for quantities

The basic question remains: Which quantity to pick for $\varphi$? In turbulent flows the turbulent kinetic energy or production of kinetic energy might be important, whereas in combustion density or temperature might be an important quantity. In this study several different variables have been tried, like the magnitude of the velocity vector, the turbulent kinetic energy, the eddy viscosity and the mixture fraction.

4.7.4 Coupling the local grid refinement to the domain decomposition

One of the restrictions of the domain decomposition is that all the blocks have to be rectangular. This poses a severe problem in the refinement, because in general the region that will be selected for refinement will not have a cubical shape. In case of turbulent combustion, using the temperature refinement criterion, the refined domain will have the shape of the flame. Thompson and Ferziger (1989) suggest to set a threshold value and mark all cells with a refinement value higher than this threshold for refinement. Next, the cells are clustered in groups. A point is added to a group if it is within a certain distance of any other point within the group. Once this clustering is complete, boundary-aligned rectangles enclosing all grid points of a
group are drawn. Finally, checks for overlapping of blocks are performed, and if they occur the overlapping blocks are split in non-overlapping blocks. This method can be implemented, but might lead to a uncontrolled, large amount of blocks. Hence we came up with two somewhat more straightforward methods, in which it is very easy to control the number of new blocks in the decomposition.

Determining the new domain decomposition

![Diagram](image)

**Figure 4.19:** Example of the new domain decomposition obtained with adaptive local grid refinement using A priori set of block boundaries. The shaded area's are the blocks where refinement is suggested. The new block decomposition is drawn in solid lines, the possible decomposition in dashed lines. The numbers denote the old and new blocks. (a) Initial computation with one block and possible block boundaries for subsequent decompositions. (b) Blocks marked for refinement (shaded). (c) New block decomposition. (d) Second computation with 7 blocks and two levels of refinement.

A priori set possible block boundaries In this case the computational domain is initially split into a number of blocks, predefined in shape, as shown in figure 4.19(a). In this figure a division of $4 \times 4$ blocks is used. Then the computation is started and the error in every node is computed, according to any of the criteria mentioned above.
The average error in a block is obtained by summation over all nodes in a block and scaling by the number of nodes in the block. This determines an error value for every of the 16 possible new blocks. The refinement level of the blocks with an error value higher than some predefined threshold value is increased. These blocks are shown in figure 4.19(b) as blocks with shaded area's. Finally the blocks are merged where possible. We have adopted a merge-algorithm in which the block stays as 'square' as possible, but other options are feasible. This merging of the blocks also determines the final number of new blocks to restart the computations and the load imbalance in the new computation in case of parallel computing. The final decomposition is shown in figure 4.19(c). A new computation starts, with a new number of blocks and new levels of refinement. Since two adjacent blocks can only differ 1 level of refinement (see section 4.4.1) the need for refinement in a particular block might cause adjacent blocks to be refined too. This procedure can be repeated as often as necessary (4.19(d)). The maximum number of blocks is always determined by the user and not by the algorithm.

Let program set new block boundaries The weak point of the above method is that the user has to specify the possible new block boundaries before the computation starts. This suggests that the user should already have some clue about refinement regions, which is exactly what you don't want to input. Hence the above algorithm can be improved by letting the program itself define the location of the possible block boundaries. The errors in all nodes are integrated over all coordinate directions: $E(x) = \int_{y,z} e(x,y,z)dydz$, $E(y) = \int_{x,z} e(x,y,z)dxdz$, $E(z) = \int_{x,y} e(x,y,z)dxdy$, and the new block boundaries are determined from maximum values of $E$. From here on the procedure is similar as mentioned above.

Impact on the new domain decomposition

To see the impact on the new domain decomposition we used the lid-driven cavity flow as suggested by Thompson and Ferziger (1989). This case was also used by Muzafarija and Gosman (1997a). Figure 4.20 shows the different grid refinements after two levels of refinement starting with a uniform $8^2$ grid using criteria b, c and d from the previous section and using the magnitude of the velocity vector as refinement quantity. Four blocks were allowed in each direction.

The results for refinement criterion c are in excellent agreement with those of Muzafarija and Gosman (1997a) and Thompson and Ferziger (1989). This method can be seen as a coarse approximation to Richardson extrapolation. The total amount of grid points differs significantly for the three criteria, as shown in table 4.5. The blockwise refinement leads to significant load imbalance for parallel computing (which will be discussed in chapter 5). The smallest block only contains 4 points. Of course this case is quite pathetic, but this load imbalance was also observed in other test-cases. For decent load balancing you need at least locally unstructured grids. Hence,
Figure 4.20: Grids, obtained with different local refinement criteria (a) Vectorplot of field for lid-driven cavity. (b) Criterion maximum curvature. (c) Criterion higher order fluxes. (d) Criterion maximum gradient.

<table>
<thead>
<tr>
<th>Criterion</th>
<th># blocks</th>
<th>Max</th>
<th>Min</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Muzaferija</td>
<td>6</td>
<td>128</td>
<td>4</td>
<td>220</td>
</tr>
<tr>
<td>Gradient</td>
<td>7</td>
<td>256</td>
<td>4</td>
<td>292</td>
</tr>
<tr>
<td>Curvature</td>
<td>7</td>
<td>128</td>
<td>8</td>
<td>376</td>
</tr>
</tbody>
</table>

Table 4.5: Number of block and nodes arising from different refinement criteria.
no further computations were done with different refinements using adaptive local grid refinement for the more complex cases. Instead, attention was focused on the prediction of the refinement criteria.

4.8 Results of local grid refinement criteria

4.8.1 The influence of local grid refinement

For the laminar lid-driven cavity flow employed in the previous section we can check the performance of the local grid refinement quite easily, since we can compare the solution against the exact solution. The global average error (defined as the average error of all quantities in all gridpoints) was computed according to equation 4.9. The grid refinement was then applied, allowing one level of refinement only, and starting with a possible 4 by 4 block distribution. It can be seen that the order of the method

![Graph showing the global average error with and without local grid refinement](image)

**Figure 4.21**: Global average error with and without local grid refinement

is approximately second order. For very coarse grids the truncation errors slightly influence the results. The local grid refinement does indeed work as expected. If we use
one level of refinement the global average error for a given number of nodes is smaller than without local grid refinement. Here again the truncation errors slightly influence results on very coarse grids, but it shows that the grid refinement in principle works. For the remaining cases no accuracy-studies were done, because of the severe load imbalance. We have looked at the performance of the different refinement criteria.

4.8.2 The influence of criteria on the local grid refinement

Backstep flow

For this case only the curvature and gradient criterion were used, for reasons mentioned in the previous section. The kinetic energy and eddy viscosity were used for refinement criteria. The resulting suggested refinement regions and the fields themselves are shown in figures 4.22 and 4.23. It can be seen that both quantities give

Kinetic energy field

Error criterion gradient, kinetic energy

Error criterion curvature, kinetic energy

**Figure 4.22**: Suggested refinement regions for the backstep flow using gradient and curvature criteria on variable kinetic energy
Eddy viscosity field

Error criterion gradient, eddy viscosity

Error criterion curvature, eddy viscosity

**Figure 4.23:** Suggested refinement regions for the backstep flow using gradient and curvature criteria on variable eddy viscosity

similar results for both gradient and curvature, mostly because the fields themselves also look quite similar, as the eddy viscosity scales as the square of the turbulent kinetic energy. The shear layer is captured quite well in all cases.

**Furnace simulations**

The refinement criteria were applied to the complete furnace computation with a gridsize of $32 \times 50 \times 50$. The results are shown in figure 4.24 and 4.25 for two quantities, namely the velocity vector and the mean mixture fraction, using the gradient and curvature of the variables as criteria. We have shown isosurfaces of the error, as this was thought to give a better indication of the distribution of the error. The block-interfaces are visible in all figures. This has to do with the fact that the error was not computed on the block boundaries, and hence set to zero there.

For the velocity the differences between the two criteria is significant. The recir-
Curvature of velocity vector

Figure 4.24: Suggested refinement regions for the furnace simulation using gradient and curvature criteria on the magnitude of the velocity.

culation zone, in the top of the furnace, gives rise to a very high curvature. It must be said that velocities in these regions are quite small and hence this criteria is not too appropriate in this case. The gradient of the velocity vectors shows some agreement to the gradient of the mean mixture fraction. However, is is much smaller.

The mean mixture fraction criteria show more agreement. The flame zone is captured quite well by both criteria. Also the air-inlet, where the hot air enters the combustion chamber, gives rise to steep gradients in mixture fraction. However, this region is not easily captured by a set of rectangular blocks. Again we see the
Figure 4.25: Suggested refinement regions for the furnace simulation using gradient and curvature criteria on the mean mixture fraction.

shortcomings for applying adaptive grid refinement for block-structured grids.

We can conclude that in general it is impossible to find an optimum set of quantities and criteria which will give similar results. Some quantities typically reach a maximum where other quantities have a maximum gradient (i.e. kinetic energy and mean velocity). A developer should always be aware of where to expect the 'interesting regions' in the flow and use the quantity that has high gradients there for refinement criteria. The mean flow quantities and gradients might serve as a good start.
4.9 Conclusions

In this chapter a general domain decomposition technique on block-structured grids has been given. The blocks can be coupled in an arbitrary manner, provided that the refinement ratio between two adjacent blocks does not differ more than a factor of two. The communication between the blocks is defined in a general way to allow for easy parallelisation and local grid refinement.

The difference between additive and multiplicative Schwartz have been addressed. Only the additive Schwartz method is suited for parallelisation, but this method is approximately twice as slow as multiplicative Schwartz. Several algorithms have been implemented to update the values on the internal block-interfaces. The flux-conservation method converges faster than the straightforward update mechanism, because of the stronger coupling. But the method is not particularly suited for fast and accurate solving of the subdomains since it quickly becomes unstable when more accurate solving of the systems is applied. Hence it was decided to use the straightforward update for all simulations. It is concluded that accurate solving for multi-block computations is not always the best way to obtain a numerically efficient code, because of the steep gradients possibly arising on the interfaces.

The convergence behaviour deteriorates when more blocks are used in a problem with a fixed size. This is caused by the fact that the equations are elliptical and by the fact that an implicit iterative procedure is used to solve the systems. The term numerical efficiency has been introduced to quantify the relative increase in iterations for multi-block computations. The decrease in numerical efficiency, which was thought to be very severe, turned out to be marginal, and only slightly problem dependent. It was shown that the decomposition into strips rather than blocks has a poor effect on the numerical efficiency and on the CPU-time.

To improve the coupling between the blocks two additional iteration loops have been implemented. It was shown that increasing the number of iterations within the solver is not very useful and only leads to slightly improved convergence behaviour. When, on the other hand, the systems are solved very inaccurately, but the halo-cells are updated after this, the matrix is rebuilt and the source terms are re-evaluated, significant improvement in convergence behaviour is obtained. The improvement is very stable and works as expected on all considered cases.

To improve the numerical accuracy, the possibility of adaptive local grid refinement was investigated. However, structured solvers only offer very limited advantage for local grid refinement, as the refinement regions have to be mapped onto the blocks to save the banded structure of the system, which will lead to high load imbalance. At least locally unstructured grids should be applied, also from a topological point of view. We have shown that it is impossible to develop a set of criteria and quantities that give reliable local grid refinement criteria. The developer will always have to put in some pre-knowledge about the characteristics of the flow for proper refinement regions.
5 Parallelisation

5.1 Introduction

The second objective of this thesis is to extend the domain decomposition technique to applications on parallel platforms. On massively parallel machines CPU and memory restrictions do not limit the size of the simulations anymore, and computations can be done using much larger gridsizes. We have performed computations on grids that contain 14 times more nodes than previous work by Boerstoel (1997). The finest grid contains almost 2 million nodes and is shown in figure 5.1. This extension to

Figure 5.1: Finest global grid used in computations of the IFRF furnace simulations. The grid consists of $97 \times 202 \times 100 = 1959400$ nodes. This grid was divided into 64 blocks.

parallelisation involves the implementation of communication routines between the blocks, which are now running on different Processing Elements (PEs). Furthermore
the I/O has to be modified, and each PE has to know which block to compute.

First we give a concise overview of the hardware and software available to build and execute parallel programs, and some considerations on the possible gains of parallel computing. Then the parallelisation of the code is discussed in detail. This approach is very general and has also been used successfully to parallelise other CFD codes. Parallelisation of the radiative heat transfer will be discussed. The relative gain of using more processors for a simulation ('parallel efficiency') is discussed. Finally we show that the load-imbalance is an underestimated problem in CFD computations on machines with a fast network.

Almost all parallel simulations were performed on the CRAY T3E/600, available at HPaC, the High Performance Applied Computing centre at the Delft University of Technology. This machine became available in 1997 and is currently one of the fastest machines available. Attention was hence focused on this machine, although the code will run on any parallel platform. Whenever we refer to the CRAY T3E, we refer to the 600-type version of the machine.

**History of parallel computing**

The term 'parallel computing' refers to performing computational tasks concurrently. The aim of parallel computing is to decrease the total wall-clock time to perform a numerical simulation.

The basic idea behind parallelisation is simple, and indeed not novel at all. It was in 1821 that Charles Babbage invented a 'Difference Engine' which could perform one addition or subtraction per second (multiplications and divisions were estimated to take a minute). He realised that the simultaneous computing of numbers was possible, but not within reach of the technological requirements of that era. Menabrea (1842) however, recognised that for identical computations, such as found in the formation of numerical tables, Babbage's machine could be used to produce several results concurrently, making it the first parallel computer. A good overview of the evolution of parallel computers can be found in Emerson (1996).

In the last decade, considerable efforts have been spent in porting CFD codes to parallel platforms (Sciano et al., 1997). Parallel programming is said to be the only computational answer for speeding up or making feasible numerical computations of complex processes, such as, for instance, encountered in industrial glass melting furnaces. These require advanced models of turbulence, combustion and radiation in conjunction with sufficiently fine numerical grids to resolve important small scale interactions in the areas of flame front, high shear and near solid walls. Currently, because of limited computer power, most numerical simulations are performed with relatively simple models, hampering an accurate prediction. Parallel processing is regarded nowadays as the promising route by which to achieve desired accuracy with acceptable turn-around time.

Alternative options would be to use even faster sequential computers or to increase the accuracy of numerical schemes. Veldman and Verstappen (1997) have tried the
latter approach and indeed gained orders of magnitude in computing time, because less iterations are needed. However, these higher-order schemes can very well be used in conjunction with parallelisation. In a separate work efforts are currently undertaken to reveal the relative gain of parallelisation, and the use of higher-order schemes with respect to CPU time is investigated. Sequential computers will eventually be limited by physical restrictions, such as the speed of light, in their maximum speed. Therefore, although PEs are still getting faster, it is highly unlikely that these sequential computers will be able to gain orders of magnitudes, which are needed to do high performance computing. A good overview of general aspects on high-performance computing is given by Van der Steen (1995).

5.2 Differences between parallel and sequential computing

5.2.1 Hardware differences

One of the largest differences between conventional (sequential) computing and parallel computing is the observation that the influence of the hardware on which the simulation is done is important for the performance of the code. To some extent this is also true for the classical sequential computing concept, but the general rule ("the faster the processor, the quicker the simulation") simply does not have to apply always on parallel platforms, because the connection between one PE and another PE and the connection between a PE and the memory becomes a crucial factor in the speed with which a program is executed. Flynn (1972) introduced a classification of computers based on the relation of instruction- and data streams, ranging from SISD (Single Instruction stream operation on a Single Data stream, the classical sequential scalar computers) to MIMD (Multiple ... — the modern multiprocessor configurations). Although his classification is still valid, the taxonomy of parallel computers is now usually based on the way the PEs are connected to the memory and to each other. Hence we will first discuss the possible ways to connect the PEs to the memory, and then the possible ways to connect the PEs to each other.

a. Memory connections

One has to determine how the processing elements (PEs) are connected to the memory that does not reside on the chip. The two most common options are shown in figure 5.2: The Shared-memory option and the Distributed-memory option. Machines, based on these architectures are consequently called Shared-memory machines and Distributed-memory machines.

a.1 Shared-memory machines  Shared-memory machines have a single address space where all the data and instructions for all the processors is stored. All processors
can access all data. This address space itself does not have to be physically on one huge chip, it can also be divided into several parts. In this case one speaks of 'virtually shared, physically distributed memory'. In the most extreme case, the memory can be located on the PEs (as with distributed memory, explained below). This is why sometimes shared memory machines are divided further into UMA (Uniform Memory Access) or NUMA (Non Uniform Memory Access), to appreciate the fact that one PE can sometimes get some data faster than another PE, although they both can access the data immediately. Examples of shared-memory machines are the HP EXEMPLAR and the CRAY C90.

The problem with shared-memory computers is that, since every PE can always change the values of all variables in the memory, the values residing in the cache of a specific PE might be outdated (cache coherency). Furthermore, as all memory access is handled by the network, the network can get congested if many PEs try to access data (memory contention or bus contention). This is said to be the main reason why future 'large' parallel computers will not be shared-memory computers.

### 5. Parallelisation

#### a.2 Distributed-memory machines

Distributed-memory machines do not share a single address space for all data and instructions. Instead, every PE has its own memory and address space. If the PE performs operations on data in its own memory, no network-access is necessary. If the data are available on other PEs, then the data are obtained via the network. Consequently, for a well-programmed application the network access is much smaller and the chances on network congestion are much smaller than with shared-memory machines. Algorithms that have to operate on distributed-memory machines can be more cumbersome to program, since a PE should know whether it has the data it needs, and if not, which PE does have the data. Examples of distributed-memory machines are clustered workstations or PC's, the IBM SP-2 and the CRAY T3E. Distributed-memory machines are still gaining popularity. A program that has been designed to run efficiently on a distributed-memory machine
is likely to run efficiently on shared-memory computers, whereas the opposite does not have to be the case. This is why we decided at the start of this study to focus on distributed-memory machines, as this will yield the most flexible and portable code.

a.3 Hybrid machines  A hybrid formulation exists, with PEs being able to access part of the memory on other PEs, but not all the data on all PEs. Examples are shared memory computers, coupled by an appropriate interconnect, such as Ethernet, ATM, LAN or WAN.

b. Inter-processor connections

The speed of which the data can be transferred to and from a PE is determined by three factors: The processor topology, the bandwidth and the latency.

b.1 Processor topology  There are numerous ways of connecting the PEs together. The most obvious connection might be a completely connected system, where every PE is connected to every other PE. However, this leads to a huge amount of connections and is not feasible for a large number of nodes. Other network topologies have come into fashion. Some of the most common ones are (Emerson, 1996) completely connected, mesh, tree, hyper cube and 3D torus topology. The various topologies differ in the maximum number of connections (which make the application faster, but the machine more expensive) and the maximum number of 'hops' (number of PEs data have to pass before arriving at their destination) needed for data to be transferred.

b.2 Bandwidth  The bandwidth describes the amount of data that can be transferred per second. This number varies from 1 to 10 Mbytes/s for an Ethernet connection between clustered workstations or PC's to 300 Mbytes/s bidirectional for the CRAY T3E. For dedicated systems (i.e. systems where the PEs and network are not shared with other users during the executing of a program) this number is relatively constant during an application. For Ethernet connection this number can vary considerably, depending on the type of interconnect and the size of the messages.

b.3 Latency  The time needed to establish the connection between two PEs is called latency. Latency is the unpredictable factor in the total time for a message to be sent, because the connection is usually established by the operating system, and depends on the other data traffic (congestion) on the network and on the physical location of the PEs in the topology. Since latency is most of the times completely hidden from a programmers point of view, parallel programmers use the phrase "Bandwidth comes from the computer vendor, latency comes from God".

The total time for a message to be sent can (approximately) be written as:

$$T_{\text{total}} = L + N/B(N)$$  \hfill (5.1)
where $L$ is the latency for the message, $N$ is the number of bytes sent and $B(N)$ is the bandwidth for this message length. For 'large' $N$ $B(N)$ is constant and optimal. For small $N$ the value becomes larger (messages might be buffered). The latency on dedicated systems is usually constant, because of the wormhole routing. For the CRAY T3E the latency is estimated to be approximately 1 ms when using the SHMEM routines, but it can be 10 times larger for PVM or MPI message passing. For networks like Ethernet connections, Bijl (1994) reports an average latency of 7ms. The fluctuations around this value are, however, much larger because of the shared resources, as explained above.

On the CRAY T3E the latency $L$ will only be important for messages smaller than 300Kb. These are typically global sums or maximum-determination. For larger messages the latency is negligible with respect to the bandwidth. For Ethernet connected machines most of these reasonings become void if the load on the system increases, as both the $B(N)$ and $L$ will be fluctuating, causing irreproducible timings.

### 5.2.2 Software differences

The second major difference between sequential and parallel computing is that for a parallel application the instructions have to be divided over the PEs. This implies three things. First of all that the compiler or the programmer should state which set of instructions should run on which machine, secondly the compiler or the programmer should provide the information on how to get the data if it is not available to the PE (for instance, when running on a distributed-memory machine). Thirdly this implies that the strict order of operation might not always be the same for every run. It is up to the programmer to specify where a strict order of operation is significant.

Some programs are more suitable for parallelisation than others. On one side of the spectre there are the algorithms that are embarrassingly parallel programs, meaning that the instructions are (almost) completely independent. The search for different number in a database for instance. On the other side of the spectre there are the inherently serial programs, where many instructions must be executed serially. Sorting of databases is such an operation. Most algorithms need to communicate at a limited number of points during execution to achieve the right answer. Three different choices can be made with respect to the parallelisation strategy for a sequential code. Automatic parallelisation, data parallel programming and message passing programming.

#### Automatic parallelisation

Automatic parallelisation, where the programmer only specifies the number of PEs to run on (i.e. as a compiler option on the command line), has not been very successful up to now. This is mainly because current compilers are not yet good enough to recognise parts of the program that can be done in parallel. With some help in the form of compiler directives this sometimes improves, but still the overhead is so high
that this usually does not lead to the desired results. Some companies do provide 'automatic parallelisation tools', which try to detect the parallelisation in the code and make it explicit for the compiler. These too are not yet mature enough to yield efficient parallel code without user intervention. Until good parallelising compilers will be available, the only way to achieve an efficient parallel algorithm is to help the compiler and the program to detect the parallelism by putting in extra calls to show the parallelisation opportunities. New languages for parallel programming are slowly gaining popularity. The most famous ones are Occam and Linda, but these are still in the development phase. Furthermore it is questionable if CFD-programmers want to rewrite the code just for parallelisation purposes.

Data parallel programming

The idea behind data parallel programming is to spread the data (the arrays) over the PEs and let every PE perform the operations that should be done on that data. The programmer should provide the way in which the arrays are distributed over the PEs. If the data, which the PE needs, resides on another PE, the compiler 'knows' this because the array layout is known, and the compiler will utter a request for the data. A particular feature of the data parallel programming model is that the programmer works with one memory address space and that only one copy of each declared variable exists in this space. This programming paradigm will lead to low-level parallelisation, meaning that typically DO-loops will be spread out over the PEs.

The difference between the sequential and parallel code is hence quite small. The programmer keeps one code but just has to spread the data over several processors. This is the most natural extension of vectorisation options or automatic parallelisation. The only drawback is that all communication is implicitly given by the array layout and handled by the compiler. Due to lack of good data parallel programming compilers the single PE performance of the resulting executable is usually slower than when the code was compiled with well-known compilers (f77, f90, c). It is difficult to really 'see' the communication between processors. In the end this is not necessary, but as compilers are still immature it is a desirable feature. The most popular data-parallel compiler is the High Performance Fortran (HPF) compiler. The Cray data-parallel Fortran dialect CRAFT yielded quite interesting results (partly because of the fairly moderate single PE performance of the resulting executable), but has been replaced by the HPF compiler too on the new CRAY T3E systems. Some of the major limitations of CRAFT were the poor single-PE performance and the fact that a program could only run on \(2^m\) nodes, \(m\) being an integer. Other data parallel compilers are Parallel Fortran (PCF -the abbreviation stands for Parallel Computing Forum) and D-Fortran. Recently, SGI has introduced openMP, but it is unclear whether this language will perform well for a variety of cases.
Message passing programming

The main disadvantage of data-parallel programming is that the communication is implicitly taken care of by the compiler. In applications where it is easy to point out where the communication is taking place in the program (as is typically the case with domain decomposition for CFD), the alternative is to implement all the communication between the processors via some message passing protocol. This approach is the most intrusive on the code, because the programmer now has to split the (natural) data structure of the problem and distribute the parts among the PEs. However, it will in general lead to the best performance on any system, as the communication is explicitly stated by the programmer. A major difference with respect to data-parallel programming is that if variable $a$ is declared in a program running on $n$ PEs, then $n$ copies of variable $a$ exist.

Message passing programming is currently the most popular parallel programming method. The reasons for this success are twofold. First of all, message passing programming is currently the only way to write efficient and portable parallel programs. And second, the Oak Ridge National Laboratory (ORNL) group provided (already in 1993) free communication libraries which could be used to write and run parallel programs on virtually all parallel computers. The free library was called Parallel Virtual Machine (PVM, Geist et al. (1994)). It was installed on all major machines, which implied that programs written using the PVM communication library would run almost everywhere, yielding a very portable code. The only disadvantage of PVM was that is was only a 'de facto' standard. Hence, it was superseded by MPI, the Message Passing Interface (Gropp et al., 1996). It is also for free (www.mcs.anl.gov/Projects/mpi/mpi.ch/index.html), and is very similar to PVM in many aspects. It has been defined as a standard.

5.3 Great expectations

5.3.1 Speedup and efficiency

For analysis of performance of parallel algorithms the most used indicators are the speedup and the (total) efficiency. They are defined as follows (Van der Vorst et al., 1991):

\[ S(n) \equiv \frac{T_{\text{seq}}}{T(n)} \quad E^{\text{tot}}(n) \equiv \frac{S(n)}{n} \]  

(5.2)

where $T_{\text{seq}}$ is the time of the sequential algorithm, $T(n)$ is the time on $n$ processors. We have taken $T_{\text{seq}} = T(1)$, although this is not completely fair, since it overlooks the effect that the algorithm is usually modified to enable parallel computing in the first place. However, in our case, the old, single-block code was only slightly modified, and the additional overhead for the parallelisation was small. The speedup gives the
raw gain with respect to the sequential case ('with 10 PEs the code runs $S(10)$ times faster'). The efficiency is a measure for the relative loss of speedup. In Schreck and Perić (1993) three factors influencing the total efficiency are mentioned. The numerical efficiency was already discussed in the last chapter. The other two are:

- Parallel efficiency. Relative loss because of the communication between PEs.

$$E_{\text{par}}(n) \equiv \frac{T_{\text{calc}}(n)}{T_{\text{calc}}(n) + T_{\text{comm}}(n)}$$ (5.3)

- Load balance efficiency. Relative idle time of processors caused by uneven load.

$$E_{\text{lb}}(n) \equiv \frac{T(n) - \min_p T_{p,n}^{\text{calc}}}{T(n)}$$ (5.4)

In the load balance efficiency the term $\min_p T_{p,n}^{\text{calc}}$, which is the total CPU-time needed by the fastest PE (as compared to the total time needed by the slowest PE $\max_p T_{p,n}^{\text{calc}} \equiv T(n)$) shows up. Here the differences in performance between PEs become visible. Schreck and Perić (1993) argue that if the number of control volumes and the number of floating point operations is equal on all PEs, and the PEs are equally fast, the load imbalance is zero. In theory this is right. However, the amount of floating point operations per second also depends on the availability of data and the cache behaviour. This means that two processors that have to perform exactly the same amount of operations might need a different amount of time for those operations. This effect will of course be quite small, but if several thousands of iterations are performed, this effect becomes more pronounced. Apart from this, when using domain decomposition, some blocks will have to do more communication than other blocks, depending on their position in the global grid. We have found that on machines with a fast network the relative time, spent in load imbalance is much higher than the relative time spent in communication, even if all PEs have the same number of nodes.

The total efficiency of a parallel code is consequently defined as:

$$E^{\text{tot}}(n) \equiv E_{\text{par}}(n) \times E_{\text{lb}}(n) \times E_{\text{num}}(n)$$ (5.5)

Amdahl's law for parallel processing

Ideally, $S(n) = n$, but in general this is not the case. The maximum attainable 'speed' of the parallel computer (namely the sum of the maximum speeds of all the PEs) is almost never attained. Amdahl (1967) made a similar observation for programs vectorised for vector processors. He found that the total speed of the program will always be limited by the fraction of the total time spent in instructions that cannot be vectorised. A similar reasoning was used by Ware (1973) to predict that the minimal wall-clock time for any parallel program will never be smaller than the time
needed to process the sequential fraction of the program. Van der Vorst et al. (1991) show tables of the expected speedup for different fractions of sequential work in a code. However, these pessimistic predictions were based on the assumption that the sequential fraction of the program is independent of the problem-size, number of iterations and number of PEs used. Moler (1987) rightfully noted that an increase in number of PEs is usually combined with an increase in problem-size and number of iterations. Hence the sequential fraction of the code will decrease as a function of the total amount of operations to be performed. This is the basis for his definition of an **efficiently parallel algorithm**, which is an algorithm for which the sequential part tends to zero when the problem size grows. Concluding, ideal speedup should be possible when the problem size is increased with the number of PEs. This kind of speedup is called **scaled-sized problems** as opposed to **fixed-size problems**, where the problem size is fixed when more PEs are added.

### 5.3.2 Single PE performance

Speedup is only half of the story, though. The total wall-clock time for an application and the speedup depend severely on the single-PE performance of the code. If a code runs very ineffectively on 1 PE, the speedup will be quite good, because the time spent in computations will be very large for a slow code. It is hence important to also ensure a reasonable single-PE performance of the code. The single-PE performance on RISC-based architectures depends, in turn, heavily on the memory access pattern. In the last decade the speed of the CPUs has grown much faster than the speed of the memory access. Typical clock-cycles\(^1\) for current CPUs are 1-10 ns, whereas the typical time for disk-access is still in the order of 10-100 clock cycles. For the CRAY \(T3E\) the clock-cycle is 3.3 ns, leading to a maximum performance of 300 MFLOPS. However, as an addition and multiplication can be done simultaneously, the maximum performance is claimed to be 600 MFLOPS.

To overcome the need for disk access for all data, a memory hierarchy exists. The fastest (and most expensive) parts of memory are the registers on a chip; these typically have an access period of 1 clock-cycle. Then most processors have a small, but also expensive, cache-memory on the chip. Usually this cache-memory is in the order of 10kb to 1Mb. The larger the cache, the greater the change on cache-hits, but the more expensive the machine. It has become popular to put more than 1 cache on the chips: A first- and second level cache. The CRAY \(T3E\) has 8kb first level cache and 96kb second level. However, most of the data have to be stored in the main memory of the PEs, or on external devices like disks or tapes. Therefore, effective use of the cache memory is of primary importance for good code behaviour. We have observed factors of 10 and more in performance loss if the cache memory was not used properly (cache thrashing). Many unexpected timing results are caused by

\(^{1}\)A clock-cycle is the time needed by a PE to execute a single instruction. Usually one speaks about the 'speed' of the PE which is the inverse of the clock-rate.
the sometimes rather unpredictable cache behaviour. Henty et al. (1997) claim that
-03,unroll2, nojump together with the enabling of the CRAY T3E data-stream buffers
optimises the single-PE performance for most CFD applications. We have used these
options too. These options enable loop unrolling and disables jumping (rather than
branching) to external functions. For the FURNACE code the following timings where
obtained on the single-PE performance on the CRAY T3E. Timings were done using
the apprentice-tool. The turbulent combustion simulation was run on the 'medium'
grid of 32 × 50 × 50 nodes. 2000 iterations were performed. The SIP(1) solver was
used, and the Nsolve parameter was also set to one. The average MFLOP rate

Table 5.1: Time spent performing different task types for the turbulent combustion
simulation on the medium grid on the CRAY T3E.

<table>
<thead>
<tr>
<th>Time in seconds</th>
<th>Relative time</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>3432</td>
<td>(43.91%)</td>
<td>executing &quot;work&quot; instructions</td>
</tr>
<tr>
<td>3476</td>
<td>(44.49%)</td>
<td>loading instruction and data caches</td>
</tr>
<tr>
<td>0</td>
<td>(0.00%)</td>
<td>waiting on shared memory operations</td>
</tr>
<tr>
<td>717</td>
<td>(9.17%)</td>
<td>waiting on MPI communication</td>
</tr>
<tr>
<td>10</td>
<td>(0.13%)</td>
<td>executing &quot;read&quot; or other input operations</td>
</tr>
<tr>
<td>1</td>
<td>(0.01%)</td>
<td>executing &quot;write&quot; or other output operations</td>
</tr>
<tr>
<td>179</td>
<td>(2.29%)</td>
<td>executing uninstrumented functions</td>
</tr>
</tbody>
</table>

for this simulation on 4 PEs was 106.96 MFLOPS. Note that the program time is
dominated by the cache- and instruction loading. Detailed analysis showed that we
had an approximate 0.8 operations per load for the cache, which is fairly low. This
low cache use is probably caused by the fact that for three-dimensional computations
it is impossible to keep all data 'close' together in the memory.

5.4 Parallelisation of domain-decomposition based codes

There are several ways to write a parallel program based on domain decomposition.
One could adopt the master-slave model, where the master program keeps track of
what should be done. Separate programs are available on the system to perform
specific tasks. As soon as the master wants to perform a task in parallel it spawns
a number of 'slaves' on the system that will do this. After that the master takes
control again. This concept is feasible for domain decomposition based codes, and
has indeed been used by Brakkee (1996) to solve the Navier-Stokes equations by
domain decomposition. However, one has to maintain all different sorts of programs
that together make up the complete code. This is, apart from a programming and
administration nuisance, quite error prone, because the communication operations
reside in different programs.
For domain decomposition the different PEs will all perform almost identical work, but on different datasets. They have to compute different parts of the domain. Apart from some very small additional I/O and some convergence checking, which has to be done by one of the PEs it is straightforward to have one program and start it several times with different grids and fields for every PE. This concept, called SPMD (Single Program, Multiple Data) was applied. It fits very nicely into the concept of domain decomposition, since it just means that the domains are mapped on the processors. The mapping is bijective, meaning that exactly one domain is computed per PE. For good load balancing it would be a desirable feature to compute several blocks on 1 PE, but that is impossible on the CRAY T3E. On clustered workstations this has been applied, but in general load balancing on workstations is difficult, as these are multi-user machines. An example of the mapping of the domains onto the processors is shown in figure 5.3. As the points in the program where communication should 

![Diagram](image)

**Figure 5.3:** Mapping of an arbitrary domain onto several processors.

take place are well defined, the message passing programming concept was adopted. The extension to parallel computing mainly consists of the replacement of the old global and local communication between the blocks by a message passing protocol.

In this study static load balancing is performed: Every domain contains (approximately) the same amount of grid points and the amount of work per grid point is assumed to be constant for all points. This is not true if the amount of work per block varies, which will occur if the number of inner iterations in the solver is not set to a preset value but determined by some residual error. The processors are assumed
to be equally fast and dedicated to the user. Otherwise unpredictable load imbalance occurs. These assumptions are not valid for example on multi-user workstation clusters. As mentioned in the introduction, attention was focused on the CRAY T3E for which this assumption is justified.

Parallel initialisation and output

During the initialisation all PEs read the topology file. They know exactly which block to compute, and to which (sub)faces of which block (and PE) they are connected. Furthermore the pressure anchor point (used to determine the absolute value of the pressure) has to be uniquely defined in the computational domain. All PEs should know in which block the anchor point resides, and what the pressure in the anchor point should be. All PEs read and write their data to separate files. This means that I/O can be done completely in parallel.

Parallel convergence criterion

The parallel program will only declare convergence if all convergence criteria are satisfied in all the blocks. The computations on a PE cannot be stopped once a required criterion has been reached in that block only. Due to the elliptic nature of the equations and the coupling of the blocks, the solution in a given block might not satisfy the new boundary conditions that stem from the updating of the halo cells by adjacent blocks. Hence, the fact that convergence is reached in a particular block merely states that the solution to the system of equations with the current boundary conditions has been found, and not that the solution to the global system is correct. Only when all PEs declare convergence the values on the interfaces will not change significantly anymore.

5.4.1 Solving radiative heat transfer in parallel FVM simulations

Radiation is not solved in a multi-block fashion, but still in the old, single-block manner. Two reasons can be given for this. First, as explained in section 2.3.3, radiation is solved as a separate integral equation over the entire computational domain at once, rather than applying the local difference operator, which is used for all other variables. Domain decomposition explicitly breaks the global domain into subdomains, assuming that the local character of the operators does not change too much. However, the integral equation requires so much information from other parts of the global domain that multi-block solving of radiative heat transfer will certainly not be effective and lead to large amounts of communication and slow convergence. Secondly, as stated in section 3.5, we use a radiation grid which is far coarser than the FV grid. Hence domain decomposition will lead to a very small number of cells per domain.
For these reasons we decided to solve the radiative heat transfer on one global grid in the parallel CFD computation. The coupling of the FV grid to the DTM grid remains as stated in section 3.5, with the addition now that one DTM cell can occupy several FVM blocks, but as there is only one DTM grid, this does not cause problems.

The solution procedure for solving radiative heat transfer in multi-block FVM simulations can be divided in three parts, which will all be explained below.

1. Mapping of required FV variables onto the DTM grid

2. Perform the ray tracing

3. Distributing the radiative power onto the FV grid

Mapping of required FV variables onto the DTM grid

All PEs have to know the geometry of the DTM grid. This can be done by making sure all PEs initially read the DTM-grid file. Then all PEs can map the required variables in their FVM block (the absorptive, emissive part of the radiative source term and the total radiative flux) on the DTM cells that coincide with their FVM block. The contributions of all PEs are summed by a global summation and the results are sent to all PEs. This maps the multi-block FVM variables onto the single block DTM grid. All PEs now have all data needed to compute the radiative power source term, and in principle all PEs can now perform the ray tracing (which is an identical operation on all PEs).

Perform the ray tracing

All PEs have all information needed to compute the radiative source term. However, the rays can be computed independently, as can be seen very quickly from the DTM method. Hence, all PEs compute a subset of the rays. The PEs will have to decide in advance which subset of rays they will trace. This has been done by dividing the total number of rays by the number of PEs and multiplying this number by the rank number of the PE in the computations. For instance, if there are \( N_r \) rays to be trace by \( N_p \) PEs, then PE \( N_t(i \leq N_p) \) will perform the ray tracing of rays \( (N_t - 1) \ast (N_r/N_p) + 1, \ldots, N_t \ast (N_r/N_p) \). The contributions of all rays are gathered on one PE, summed and sent back to all PEs. This yields the radiative source term in all of the DTM cells.

Distributing the radiative power onto the FV grid

This source term is mapped back onto the FVM grid in the same manner as described in section 3.5. Each PE has the complete DTM information and therefore the mapping of the source term onto the separate grid blocks can be done entirely in parallel.
5.5 Ensuring portability and flexibility

To maintain a portable code a flexible change from one message passing library to another was considered compulsory. This was achieved by using generic subroutines for all communication and parallel statements. Five subroutines were written which contain all machine specific statements: `parstart` and `parstop`, to start and stop a parallel program respectively, and check whether machine settings are consistent; `parsend` and `parrecv` for point-to-point communication, and `parglobal` for all global communication (global sums, global maxima, barriers, broadcasts and gathers are used currently).

All message passing calls were 'hidden' in these subroutines. This approach turned out to yield very portable code, and enables the programmer to use the fastest message passing protocols available on each specific platform. Another major advantage is that inexperienced parallel programmers can immediately use the same subroutines to extend their code to parallel (message passing) codes, enabling very fast porting of codes (Wouters, 1998) to a variety of parallel platforms. Appendix A shows the sources for the communication subroutines.

As a porting test the program was run on clustered HP and SGI workstations, on the IBM SP1, the CRAY T3D and CRAY T3E and finally on the locally available (and locally constructed) Linear Processor Array machine DEMOS, the Delft Molecular Dynamics Simulator (Bakker et al., 1995). The message passing protocols PVM and MPI were adopted on all machines, the machine-specific message-passing tool SHMEM on the CRAY T3E and the locally written message-passing tool FIFOGET/PUT on the DEMOS. However, in this study only results on the CRAY T3E using MPI will be shown, unless stated otherwise, because this machine fits best the above assumptions and is the fastest parallel machine currently available.

5.6 Timings

All timing tests were performed on the complete turbulent combustion simulation. Since it is very difficult to obtain converged solutions to the combined problem of heat and mass transfer, especially on finer grids, we first obtained the solution on the coarse mesh, interpolating this to the finer meshes to obtain a reasonable starting guess.

5.6.1 Speedup

The first indication of the parallel performance of a code is the speedup. We measured speedup for several different gridsizes. The `Nsolve` parameter was set to 1. The influence of increasing this parameter on the total CPU time is studied in section 5.6.2. One inner iteration in the SIP-solver was used. For the topology the most 'blockwise' distribution was used, since we concluded from the previous chapter that
this yields the best numerical efficiency. The influence of the topology of the domain decomposition on the speedup is discussed in section 5.7.3. The speedup timings were

<table>
<thead>
<tr>
<th># Domains</th>
<th>2D code ((Nblx \times Nbly))</th>
<th>3D code ((Nblx \times Nbly \times Nblz))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(1 \times 1)</td>
<td>(1 \times 1 \times 1)</td>
</tr>
<tr>
<td>2</td>
<td>(1 \times 2)</td>
<td>(1 \times 2 \times 1)</td>
</tr>
<tr>
<td>4</td>
<td>(2 \times 2)</td>
<td>(1 \times 2 \times 2)</td>
</tr>
<tr>
<td>8</td>
<td>(2 \times 4)</td>
<td>(2 \times 2 \times 2)</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>(2 \times 3 \times 2)</td>
</tr>
<tr>
<td>16</td>
<td>(4 \times 4)</td>
<td>(2 \times 4 \times 2)</td>
</tr>
<tr>
<td>32</td>
<td>(4 \times 8)</td>
<td>(2 \times 4 \times 4)</td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>(4 \times 4 \times 4)</td>
</tr>
</tbody>
</table>

also done for a 2D turbulent combustion code, called BIGMIX, which was parallelised in the same way, using the available subroutines from this study. Details of this code can be found in Peeters (1995). Table 5.2 shows the decompositions used for the speedup measurements. Results of the measurements are shown in figure 5.4. These are all 'fixed problems', meaning that the amount of nodes per simulation remains constant if the global domain is divided into more blocks. If more processors are used, every processor will have less work to do and consequently relatively more time is spent in communications and synchronisation, which will decrease the speedup. From 5.4(top) we see that the speedup is what is called super-linear. The code runs more than twice as fast on two PEs. This behaviour is caused by the single-PE performance. It turns out that when a parallel program is divided among processors, sometimes the amount of cache-hits (or cache misses) can change significantly. It is very difficult to actually use this behaviour for programming purposes. The smaller the amount of data (i.e. the smaller the number of nodes) on a PE, the greater the chance on good cache behaviour (note again that BIGMIX is a 2D code). Figure 5.4(bottom) shows the measured speedup for the full furnace simulations with the Furnace code. It is seen that the more work there is for a PE, the longer the speedup remains good. For the coarse and medium grid, using more than 16 PEs does not lead to an improved speedup. From these simulations we can conclude that turbulent combustion simulations can be done efficiently on parallel computers and that very good speedup can be obtained on large grids. The very fine grid contained almost two million nodes. Parallel computing has made these simulations feasible.

5.6.2 Influence of Nsolve parameter

In section 4.6.1 we showed that the stability of the multi-block code can be increased through the setting of the Nsolve parameter. The only price for this much more stable convergence behaviour is that the computing time per outer iteration increases. The
Figure 5.4: Measured speedup for the turbulent combustion simulations on different gridsizes. Top: Speedup for 2D turbulent combustion code (BIGMIX) Bottom: Speedup for 3D turbulent combustion code (Furnace)
combined effect is that we have to perform less iterations, but they are more expensive, hence it interesting to see the total CPU time for the different combinations.

![Graph showing CPU time vs. Number of Nsolve iterations for 8 and 16 blocks.](image)

**Figure 5.5**: Influence of the *Nsolve* parameter on the wall-clock time in seconds for the furnace simulations on 8 and 16 blocks with the $32 \times 50 \times 50$ grid.

Figure 5.5 shows that if a limited amount of *Nsolve* iterations is performed the total CPU time is approximately constant for the furnace computations. This means that the product of the number of outer iterations and CPU time per iteration is constant. The building of the matrix only becomes prohibitive for larger values of *Nsolve*, in which case the additional amount of iterations is not effective anymore.

Figure 5.6 shows that a similar behaviour is found for the cavity flow. Here the total wall-clock time even decreases slightly if a limited number of *Nsolve* iterations is performed, because this is a 2D problem, so the amount of computations is relatively small. However, when increasing the number of *Nsolve* cycles too much (say more than 5) the additional computational overhead becomes larger and the total CPU time starts to increase. Note also the dependency of the topology here. The decomposition into strips is less attractive from a computational point of view. The measured CPU times are always larger than with a more 'block-wise' decomposition. An explanation
5.6. Timings

Figure 5.6: Influence of the Nsolve parameter on the wall-clock time in seconds to calculate the 2D lid-driven cavity flow of Shih (1991) with FURNACE on the CRAY T3E. The global domain consisted of 64 x 64 nodes. Several decompositions are shown.
for this fact will be given in section 5.7.

5.6.3 Effect of message passing protocol on speedup

Vendors often suggest that machine-specific libraries yield the highest performance. This is only true if the communication is the bottleneck in the program, i.e., if there is a low parallel efficiency. For many CFD applications the PEs are only loosely synchronised, and the amount of messages is relatively small compared to the amount of computations. In most domain decomposition problems, the size of the boundaries scales with $N^2/P$, $N$ being the number of grid points in one direction and $P$ being the number of processors used, whereas the typical work per grid point grows as $N^3/P$. For larger grids ($N^3 > 250.000$) the communication time itself becomes relatively small, especially on machines with fast networks like the CRAY T3E. Other aspects, in particular load-imbalance, play a much more important role in those applications, as will be shown in the next section.

As the time needed for communication is not the bottleneck in our code, it is not compulsory to use the quickest message passing libraries available. A distinct reason not to use such libraries is that these are machine dependent, hence not portable, and sometimes show unpredictable behaviour, like cache incoherency or even bugs in the library. As will be shown in the next section, the total communication time is less than 1%. Of course on systems with a slow network, like workstations clustered by Ethernet this communication becomes much more of a bottleneck, but mainly because of the low bandwidth and high latency than because of the message passing protocol. In MPI it is also possible to map a specific PE grid on a domain decomposition, making sure that blocks that are physically neighbours also run on adjacent PEs. However, based on above arguments it was decided that not much can be gained from this modification therefore such a modification was not undertaken.

5.6.4 Input/Output

Programs are most efficient when the resources they need are dedicated to the user. Operations that require the help of the Operating System can take a considerable amount of CPU time, because these resources have to be shared with other users. The primary code part where the OS is needed is the reading and writing of data to and from files. In particular the opening, closing and existence-checking of files can become very time-consuming. In the original sequential code every variable could optionally be read from a different file, causing many I/O interrupts. In the modified version, all data were made available in a single data-file, and the amount of file checking was minimised. Furthermore, the reading of the chemistry lookup table was only done by one PE, and then broadcasted to all other blocks using the parglobal routine, rather than every block reading from disk.

Finally the fields were stored separately for every block. This means that every PE can now read and write data independent from every other PE in the application.
This can be done in parallel, provided the network does not get congested. The

total amount of I/O time for the initial and resulting code is depicted in figure 5.7
for a furnace computation on 16 PEs using the medium 32 x 50 x 50 grid. Per PE
approximately 25 double-precision variables were read from disk. The total size of the
lookup table was 15 Mb. The mean time spent in I/O was brought down from 197.1
seconds to 15.4 seconds. Also the differences per PE are much smaller in the final
code. The differences are caused by the fact that in the initial code much interaction
with the OS was needed. We can conclude that significant improvements can be
obtained by minimising and optimising I/O.

Also the physical location of the data on the disk is important. If data is only
available on remote systems through NFS (or similar), disk-access time will be de-
pendent on the load of the system, number of users and speed of the network, leading
to unpredictable I/O timings and severe load imbalance.

Of course I/O only takes place at the beginning and the end of the program,
and will probably only contribute marginally to the total wall-clock time. However,
optimising I/O will lead to a much more understandable behaviour and is therefore

Figure 5.7: Total time spent in I/O operations for each PE for a furnace simulation
on 16 blocks using the medium grid. The white bars show the I/O time in the initial
code, the dark bars show the I/O time in the final code.
strongly recommended.

5.7 Load imbalance

5.7.1 Performance per code part

Load imbalance occurs when one PE needs more CPU time to perform its operations than another PE. This difference in CPU time can be caused by several reasons, like a block containing more nodes than another block or having to do more communication. To assert the question of load imbalance we performed detailed timings for every block in the simulation. This gives an indication of the parts of the program in which most of the CPU time is spent. Furthermore it shows the load imbalance as a function of the block number.

We have done 500 iterations in the furnace simulation on the very fine grid of almost 2 million nodes. The Nsolve parameter was set to one. The total CPU time was 3053s. A global indication of the percentage of total CPU time spent in different code parts for a particular block in this simulation is given in table 5.3. Table 5.4 gives a more detailed overview. In the code there are 3 major parts; the building and

<table>
<thead>
<tr>
<th>Task</th>
<th>Percent of total CPU time</th>
<th>specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIALISATION</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>ITERATIONS TOTAL</td>
<td>98.5</td>
<td></td>
</tr>
<tr>
<td>SOLVED VARS.</td>
<td>78.9</td>
<td></td>
</tr>
<tr>
<td>UPDATED VARS.</td>
<td>17.2</td>
<td></td>
</tr>
<tr>
<td>OUTPUT</td>
<td>0.7</td>
<td></td>
</tr>
<tr>
<td>SUMMED</td>
<td>100.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Global indication of the percentage of total CPU time spent in different code parts for the very fine grid simulation

solving of the system and the barrier waiting time. Initialisation time is negligible with respect to the iteration part, even on this 50 minutes run. The solving of the independent variables (all variables for which a transport equation is solved) is 80% of the time. The updating of the dependent variables as density, temperature, etcetera, accounts for the remaining 20%. A detailed analysis on subroutine level for one block has been given in table 5.4. The amount of time spent in actual communication for this block is completely negligible. In total less than 1% of the total time is spent in actual communication. This involves all hardware issues and message passing library depending operations. It can be concluded that on a machine like a CRAY T3E with very fast network and large bandwidth it is completely superfluous to search for the fastest protocol available. Of course there is a dependency of these timings on the total number of nodes in the calculation. Figure 5.8 shows the average time spent in
### Table 5.4: Detailed overview of the total CPU time spent in different code parts for the very fine grid simulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Tot. time (sec)</th>
<th>% tot</th>
<th># calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATRIX COEFFICIENTS</td>
<td>692</td>
<td>23.2</td>
<td>6000</td>
</tr>
<tr>
<td>MATRIX SOLVER</td>
<td>607</td>
<td>19.9</td>
<td>6750</td>
</tr>
<tr>
<td>BARRIER WAITING TIME</td>
<td>587</td>
<td>19.3</td>
<td>64218</td>
</tr>
<tr>
<td>SOURCE TERMS</td>
<td>561</td>
<td>18.4</td>
<td>6000</td>
</tr>
<tr>
<td>RADIATION MODEL</td>
<td>147</td>
<td>4.8</td>
<td>155</td>
</tr>
<tr>
<td>BND. COND. EXTERN</td>
<td>60</td>
<td>2.0</td>
<td>752</td>
</tr>
<tr>
<td>BND. COND. INTERN</td>
<td>34</td>
<td>1.1</td>
<td>9754</td>
</tr>
<tr>
<td>PT2PT COMMUNICATIONS</td>
<td>18</td>
<td>0.6</td>
<td>117024</td>
</tr>
<tr>
<td>GLOBAL COMMUNICATIONS</td>
<td>3</td>
<td>0.1</td>
<td>2853</td>
</tr>
<tr>
<td>SUMMED</td>
<td></td>
<td>95.8</td>
<td></td>
</tr>
</tbody>
</table>

Building and solving of the systems and updating the boundaries as a function of the grid size for the same furnace simulation mentioned above. For the coarse, medium and fine grid computations the SIP(5) solver was used. Hence the time spent in the solver is higher in these cases. For the very fine grid the SIP(1) solver was used to decrease the computing time. It can be seen from figure 5.8 that when using more blocks and a coarser grid the amount of time spent in updating the internal boundaries (which requires communication and hence acts as a barrier) becomes higher. Especially for the coarse and the medium grid this effect is severe. This was also visible in the speedup curve 5.4 (b). For the larger grid the communication part becomes small.

#### 5.7.2 Timings per code part for all blocks

The question of course remains whether these average values are representative for the fluctuations in timings that actually cause the load imbalance. Therefore, figure 5.9 shows the percentage of CPU time for the very fine grid simulation mentioned above for every block in the simulation. We see that the time, needed to build and solve the matrix fluctuates slightly, but the barrier time differs much more between the blocks. The highest total barrier waiting time (the percentage of idle time for this block) is almost 20%, whereas the smallest total barrier waiting time is some 4%. Analysis showed that this imbalance stems from the fact that in our study the complex geometry is modelled with the porosity method (Moult et al., 1979), which marks the cells that lie outside the fluid domain in the Cartesian mesh as 'closed cells'. In these cells all quantities are set to a preset value, and no equation is solved for these nodes. Figure 5.10 shows the total barrier waiting time of every PE together with the number of closed cells in that block. The blocks which contain many closed cell are obviously quicker than other blocks, and hence this causes enormous load
imbalance. The best thing to do is rearrange the blocks such that every block has the same amount of 'not-closed' cells, but this would involve a quite complicated block decomposition and has therefore not yet been tried. But apart from this there is still an average load imbalance of approximately 5%. This has to be attributed to the fact that even if blocks have the same amount of points, then still the total amount of work differs. This is addressed in the next section.
5.7. Load imbalance

![Graph showing relative amount of CPU time for every PE in the very fine grid simulation for the 3 most time-consuming code parts: the building and updating of the matrix and the barrier waiting time.](image)

**Figure 5.9:** Relative amount of CPU time for every PE in the very fine grid simulation for the 3 most time-consuming code parts; the building and updating of the matrix and the barrier waiting time.

### 5.7.3 Effect of block decomposition on load imbalance

To show how the block decomposition itself can cause load imbalance the coarse grid was split into four blocks. Two configurations were studied, shown in figure 5.11. In the left decomposition all blocks have exactly two neighbours, in the right one the middle two blocks have two neighbours, and the outer blocks one. Every block contains the same amount of points for both decompositions. The timings for both runs are depicted in table 5.5. The rectangular decomposition yields much better timing results than the sliced decomposition. This is due to the fact that in the sliced decomposition, the two middle blocks have twice as much communications to do, creating an huge load-imbalance in that part of the code. This effect becomes smaller if the number of points per blocks becomes bigger. The total communication time is approximately equal in both cases, and is small compared to the entire program time. The smaller parallel working time in the rectangular decomposition is because of better single PE-performance. This is related to the shape of the blocks and
Figure 5.10: Relation between barrier waiting time and the number of cells in which the solution is not computed (closed cells) for the very fine grid simulation

<table>
<thead>
<tr>
<th>Time needed (s)</th>
<th>sliced decomp.</th>
<th>rectangular decomp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total program time</td>
<td>2150</td>
<td>1700</td>
</tr>
<tr>
<td>Parallel working time</td>
<td>1140 (53%)</td>
<td>1241 (73%)</td>
</tr>
<tr>
<td>Total barrier waiting time</td>
<td>606 (28%)</td>
<td>106 (6%)</td>
</tr>
<tr>
<td>Total communication time</td>
<td>85 (4%)</td>
<td>77 (5%)</td>
</tr>
</tbody>
</table>

Table 5.5: Difference in time spent in various program parts for the furnace computation on the coarse grid. 4 blocks were used with rectangular (2x1x2) and sliced (1x4x1) decompositions

consequently the size of the arrays. The rectangular decomposition yields blocks with longer interfaces; their volume to surface ratio is slightly smaller. This, in turn, leads to slightly more communication per iteration. This effect is seen to result in only 1% more communication time, since the global grid is quite small.
Figure 5.11: Two decompositions of the same coarse grid. Left: Rectangular decomposition. Right: Sliced decomposition.

5.7.4 Other reasons for load imbalance

There are many other reasons for load imbalance. Verweij et al. (1999) quotes some more reasons.

5.8 Conclusions

Domain decomposition proves to be an easy way to port a sequential code to a parallel platform using message passing paradigms. Generic communication subroutines have been written to enable easy porting of the code to other platforms. These subroutines have proven to be general enough to be used in other codes too.

In our case the total time spent in actual communication was found to be less than 1% for all computations on the CRAY T3E. This is the consequence of the very fast network on this machine. Consequently, use of machine specific message passing libraries will not lead to a much faster code, as communication time is not the bottleneck in the code and so the use of portable libraries, such as MPI is advised.

The most cumbersome part of the parallelisation is the load imbalance. This turns out to be more intrusive than expected. Load imbalance can be only partially attributed to the fact that the load is not completely even on all PEs. Also the number of adjacent blocks influences the load imbalance. The performance of a code can be improved significantly if domains all have the same number of neighbours (up to 8 blocks), or if the updating can be done in the same time for all blocks. The size of the subdomain interfaces does not influence the parallel performance if the latency
and bandwidth are reasonable. Cache use is very important if one wants to obtain good single PE performance on the CRAY T3E.

**Future of parallel computing**

It is difficult to say what the future of parallel computing is. Slowly but surely people understand that codes should run in parallel to really get rid of CPU and memory limitations. However, engineers still see parallel programming as an annoying programming habit. They are right in this respect. I think the future of parallel programming lies in parallelising compilers, such that running in parallel would just require a compiler flag. Only then people will start using parallel computers on a large scale. However, as much of the parallel performance still depends on the hardware, the state of the art in good, or even decent parallelising compilers or automatic parallelisation tools is still very low, mainly because such a high burden is placed on the compiler. Single PE performance of these compilers has to improve, and they should get more stable. Until then a speaker I once saw on a conference in 1994 will be right when he said: 'Parallel computing is the technology of the future', and added, '... and always will be.'
6 Furnace simulations

6.1 Introduction

The previous chapters focused on the mathematical and computational aspects of the simulations. The implementation of the domain decomposition and the parallelisation have extended the realm of the computations we want to perform. The restrictions with respect to CPU-time and memory have been completely overcome in this study. The grids can have arbitrary size and can be run in reasonable turn-around time.

However, in the end all these investigations have been employed to perform more accurate simulations and to apply better models. Therefore, the last result chapter focuses on the extended applicability of the new code. All the methods explained and tested in the previous chapters are merged, showing how the features added in this study have extended the possibilities for furnace computations.

All computations discussed in this chapter are performed with the parallelised code using domain decomposition. Only the IFRF-furnace computations are considered. These furnace simulations were performed with and without local grid refinement. Section 6.3 shows results without local grid refinement. These results can be directly compared to the results obtained by Boerstoel (1997) and to the measurement done during the NG7-trials, as explained in section 3.6. Section 6.4 finally discusses the results obtained with local grid refinement. This section hence exploits all possibilities considered in this study; parallel furnace simulations using domain decomposition with local grid refinement.

6.2 Details of the computations

The IFRF-furnace computations were carried out for the underport flame only, using a gas inlet velocity of 125 m/s and a gas injection angle of $\alpha_{gas} = 16^\circ$. Most details with respect to the boundary conditions have already been discussed in section 2.4 and will not be repeated here. For all computations a uniform radiation grid of size $4 \times 20 \times 6$ with 16 beams was used as this is the same radiation grid that was used by Boerstoel (1997). No efforts were undertaken to investigate the influence of
the radiation grid on the solution, since Boerstoel (1997) already carried out these computations. No soot models and no NO\textsubscript{x} models were present in the code. This means that the temperature predictions will be slightly higher than in the simulations Boerstoel uses. At the start of this study, implementation of the NO\textsubscript{x} simulations in the code was deferred, as we wanted to study the effect of block decomposition and parallelisation on the computations. NO\textsubscript{x} is solved by a post-processor, and hence does not significantly alter these conclusions. Within the framework of this study it was not possible to re-include the NO\textsubscript{x} simulations in the code, due to lack of time.

6.3 Computations without local grid refinement

This section focuses on the results of the furnace simulations without the use of local grid refinement. The parallel multi-block code is used to show that the results obtained can be related to the results of Boerstoel (1997), and that almost grid independent results were obtained. The grids used for the computations without local grid refinement were quoted in table 3.9 already, but are repeated in table 6.1 for clarity. In this table we also denoted the grids used by Boerstoel (1997) for the

<table>
<thead>
<tr>
<th>Mesh</th>
<th>(N_x \times N_y \times N_z)</th>
<th>#CVs</th>
<th>Boerstoel (1997)</th>
<th>(N_x \times N_y \times N_z)</th>
<th>#CVs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse</td>
<td>16 \times 24 \times 20</td>
<td>7680</td>
<td>16 \times 24 \times 20</td>
<td>7680</td>
<td></td>
</tr>
<tr>
<td>Medium*</td>
<td>32 \times 50 \times 50</td>
<td>80000</td>
<td>24 \times 36 \times 30</td>
<td>25920</td>
<td></td>
</tr>
<tr>
<td>Fine</td>
<td>62 \times 98 \times 82</td>
<td>498232</td>
<td>32 \times 48 \times 40</td>
<td>61440</td>
<td></td>
</tr>
<tr>
<td>Very fine</td>
<td>97 \times 202 \times 100</td>
<td>1959400</td>
<td>32 \times 72 \times 60</td>
<td>138240</td>
<td></td>
</tr>
</tbody>
</table>

* Boerstoel calls this the 'standard' grid.

simulations of the iFRF-furnace; He performed similar computations, but was not able to reach the same level of grid fineness. He used the same terms 'coarse, medium, fine and very fine' to distinguish between grid sizes. However, the actual sizes of the grid differ significantly, as can be seen from the fourth and fifth column of table 6.1. This increase in grid-nodes is feasible through the use of many blocks on a dedicated parallel machine. For these simulations the coarse and medium grid calculations were done on 1 respectively 1 and 8 blocks. The fine grid simulations were done on 16 blocks. The very fine grid simulations consisted of 64 blocks, using almost the entire capacity of the CRAY T3E. These grids show that the previous restrictions with respect to CPU-time and memory have been completely overcome in this study. The grids can have arbitrary size (as shown here) and can be run in reasonable turn-around time (as shown in the previous chapter). The grid node distribution in the \(y - z\) plane
Figure 6.1: The grids used for the furnace computations. Shown is the grid in the $y - z$ plane.
(the symmetry plane) has been plotted in figure 6.1. The refinement around the air- and gas-inlet (in the left of the grids) is clearly visible. Figures 3.6 and 5.1 might be helpful to get a three-dimensional understanding of the grid-layout. The unnecessary refinements in all other parts of the furnace is also visible. However, the finer grids did have better cell aspect-ratio's, which led in practise to more stable convergence behaviour.

The mean temperature field in the symmetry plane is shown in figure 6.2 for several grid sizes. This figure also shows a close-up of the temperature field around the inlet section. Figure 6.3 shows a close-up of the mixture-fraction and the mixture-fraction variance near the gas-inlet. The inconsistencies along the block interfaces are caused by errors in the visualisation package. In the code itself all quantities are continuous.

![Field for coarse grid](image)

![Close-up field for coarse grid](image)

**Figure 6.2:** Temperature field, together with close-up near the gas-inlet for different grid sizes. Isolines are shown for 1300(100)2200 K.
Figure 6.2: (cont.) Temperature field, together with close-up near the gas-inlet for different grid sizes. Isolines are shown for 1300(100)2200 K
Figure 6.3: Close up of mixture-fraction (left) and mixture-fraction variance (right) field near gas inlet for different grid sizes. **Top** Coarse grid. **Middle** Medium grid. **Bottom** Fine grid.
along block interfaces. The predicted temperature profile improve considerably with increasing grid-size. The steep gradients near the inlet sections are better captured by the finer grids. Also the thickness of the flame-front decreases, showing a smaller amount of numerical diffusion. Figure 6.3 shows that the behaviour of f and g is good.

A more detailed comparison of the temperature-predictions as a function of the grid-size is given in table 6.2. Also the temperature prediction by Boerstoel (1997) is included. It should be noted that the simulations of Boerstoel were done with a staggered version of the code, whereas the current computations have been done with the colocated version; it was shown already in figure 3.2 and table 3.1 that this leads to somewhat higher temperatures, mainly because of the improved treatment of the inlet boundary conditions.

The temperature profiles have also been measured for the in-flame calculations during the NG7-trials, as explained in section 3.6. These measurement were already discussed by Boerstoel (1997). Figure 6.4, courtesy of Boerstoel (1997), shows the points of measurements, together with the predicted temperature field in the symmetry plane, that followed from his simulations. He used these measurement to compare the computations, but only for his 'Fine grid' simulations. Figure 6.5 shows the predicted temperature fields for different grid sizes at several measured locations as a

**Table 6.2: Computed maximum and averaged temperature in the furnace.**

<table>
<thead>
<tr>
<th>Mesh</th>
<th>Maximum temperature (K)</th>
<th>Averaged temperature (K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse</td>
<td>2502</td>
<td>1717</td>
</tr>
<tr>
<td>Medium</td>
<td>2529</td>
<td>1732</td>
</tr>
<tr>
<td>Fine</td>
<td>2538</td>
<td>1723</td>
</tr>
<tr>
<td>Very fine</td>
<td>2652</td>
<td>1708</td>
</tr>
<tr>
<td>Boerstoel (1997)</td>
<td></td>
<td>1700</td>
</tr>
</tbody>
</table>

**Figure 6.4:** Results of simulations of the IFRF UP, 125 m/s, 16° flame in the symmetry plane of the furnace. The points of measurement are indicated by a black dot. (Figure taken from Boerstoel (1997)).
Figure 6.5: Comparison between simulations and in-flame measurements. X is the distance from the symmetry plane, Y is the distance from the inlet. Shown are the temperature profiles along the height of the furnace for the coarse grid (solid line), medium grid (dotted line) and fine grid (dashed line). Cubes are measurements.
function of the height of the furnace. The measured data are shown too. The markers on the lines show the locations of the computational nodes. The differences between the grids are apparent. The coarse and medium grid offer too little refinement in the flame zone, and do not capture the nearly symmetric jet very well, which is clearly visible at \( x = 0.0 \text{ m} \), \( y = 0.6 \text{ m} \). The fine grid predictions agree very well with the measurements, especially in the upper part of the furnace, in the recirculation zone. Near the outlet part of the furnace, the predicted temperatures are somewhat lower than the measured values. Boerstoel (1997) argues that this might be attributed to buoyancy effects. The slight overprediction near the inlet is caused by the fact that we do not take into account the soot-formation in the furnace. The presence of soot will decrease the heat release and increase the radiation and hence lower the flame temperature. Near the furnace exit, the soot will have been combusted again and only a small amount of soot will be present in the flow. Here the measurement agree better with the predictions indeed.

**Visualisation of large datasets**

One problem with the visualisation is that it turned out to be extremely difficult to analyse the data if the number of blocks and the grid size increases. In practise most packages are restricted to run on workstations, and cannot process the vast amount of data produced by the simulations. Several options were investigated to overcome this problem. First we tried to interpolate the data on a single block mesh which did fit into a workstation memory. However, this led to interpolation errors and loss of detail. Another option was to let the FURNACE-program write only some of the interesting variables instead of all variables. This meant intervening in the code itself and would mean rerunning the code when additional variables were required for analysis. Finally we decided to write a complete conversion program that could extract a user-defined portion of the computational domain out of the complete dataset. This led to satisfactory visualisation, but at the same time shows the genuine bottleneck that still exists for the visualisation of large datasets. Some work has been undertaken into parallel visualisation and postprocessing by e.g. Haines (1997) and Rantzau and Thomas (1997) and consists of parallelisation of the visualisation tools themselves. Haines (1997) designed a tool called pV3 (parallel Visualisation 3), running under PVM and MPI. However, most commercial tools still have a long way to go before they are able to cope with these large datasets.

### 6.4 Computations with local grid refinement

In this last section the influence of local grid refinement on the simulations is shown. This tool intends to give the user more influence on the locations of the nodes. The adaptive local grid refinement, with the code deciding where to refine the grid, was
Figure 6.6: a: Global view of the decomposition into 6 blocks with local grid refinement in some of the blocks. b: Front view. c: Side view. d: Top view.
6.4. Computations with local grid refinement

not applied, for reasons explained in the previous chapter. Instead, the user has to provide:

1. The dimension of a uniform, global (coarse) grid

2. A table in which the number of block with topological position and refinement level is stated.

The global grid used for this computation consisted of $16 \times 24 \times 20 = 7680$ nodes, so the same number of nodes as in the 'coarse grid' simulation. These 6 blocks were divided as quoted in table 6.3. The total number of nodes is now 50304, which is even less than

**Table 6.3**: Geometrical position and number of nodes for the blocks in the furnace simulation with local grid refinement

<table>
<thead>
<tr>
<th>Block</th>
<th>X-size</th>
<th>Y-size</th>
<th>Z-size</th>
<th>refinement level</th>
<th>Resulting #CVs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00-0.44</td>
<td>1.53-4.09</td>
<td>0.00-0.96</td>
<td>1</td>
<td>$16 \times 15 \times 20 = 4800$</td>
</tr>
<tr>
<td>2</td>
<td>0.00-0.44</td>
<td>1.02-1.53</td>
<td>0.00-0.96</td>
<td>2</td>
<td>$32 \times 6 \times 40 = 7680$</td>
</tr>
<tr>
<td>3</td>
<td>0.00-0.44</td>
<td>0.00-1.02</td>
<td>0.00-0.09</td>
<td>2</td>
<td>$32 \times 12 \times 4 = 1536$</td>
</tr>
<tr>
<td>4</td>
<td>0.00-0.11</td>
<td>0.00-1.02</td>
<td>0.09-0.62</td>
<td>3</td>
<td>$16 \times 24 \times 44 = 16896$</td>
</tr>
<tr>
<td>5</td>
<td>0.00-0.44</td>
<td>0.00-1.02</td>
<td>0.62-0.96</td>
<td>2</td>
<td>$32 \times 12 \times 14 = 5376$</td>
</tr>
<tr>
<td>6</td>
<td>0.11-0.44</td>
<td>0.00-1.02</td>
<td>0.09-0.62</td>
<td>2</td>
<td>$24 \times 12 \times 22 = 6336$</td>
</tr>
</tbody>
</table>

the number of nodes in the 'medium grid' without local grid refinement. Figure 6.6 shows the blocks and the resulting grid from three different views. We will call this the LGR grid for the remainder of this section. The global grid in these computations is now uniform, rather than stretched, as used in the previous computations. This leads to a much more smooth convergence behaviour. In the previous section we already noted that a decent aspect-ratio is very important for good convergence behaviour and improves the numerical accuracy. We found that the computations with local grid refinement with an underlying uniform grid all converged smoothly. In the coarse grid-computations of the last section we found that solving all variables (especially density) from scratch often led to divergence. In this case none of these problems were encountered. The load-imbalance in this case is quite high. Block 3 contains only 10% of the nodes of block number 4. For parallel efficiency it would be advisable to either split block 4 in several smaller blocks, or increase the size of block 3. Here we see the complication of local grid refinement combined with parallelisation. Part of this complication can be relieved by computing several blocks per PE, but this would require a more severe programming effort and was not undertaken. Unstructured grids yield more parallel-efficient multi-block codes, as the number of cells in each
block can be changed easily. However, in this simulation we are merely concerned with the predictions rather than the parallel efficiency.

The simulated mean temperature field for the LGR grid at the symmetry plane is shown in figure 6.7. The gradient of the temperature at the inlet section is better cap-

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{image.png}
\caption{Temperature field with the locally refined grid. Top Entire domain. Bottom Close-up near the gas inlet. Isolines are shown for 1300(100)2200 K.}
\end{figure}

tured by this grid than by the medium grid from the last section. The inconsistency along the block interfaces is again caused by the visualisation package. The predicted maximum- and average temperature in this computations were 2327 K respectively 1698 K. The maximum temperature is hence significantly lower than in the previous computations. The mean temperature however, agrees very well with the mean temperature for the very fine grid simulation. Also the temperature increase above the air inlet, caused by the recirculation of remaining fuel, is much more visible. Figure 6.8 shows the predicted mixture-fraction and mixture-fraction variance profiles around the inlet section. The mixture-fraction variance profile extends much more into the flame zone than with the previous computations. This might be caused by the fact
that the mixture-fraction gradients are much better captured by the finer grid near the inlet, although it is not visible in the fine grid simulations of the last section.

![Figure 6.8](image.png)

**Figure 6.8:** Close up of mixture-fraction (left) and mixture-fraction variance (right) field near gas inlet for the LGR grid.

Detailed comparison is done again by comparing the results to the experiment, just as in the previous section. Figure 6.9 shows the temperature field along the height of the furnace at several locations. The results of the medium and the fine grid from the last section have been plotted too. The medium grid has approximately the same amount of nodes as this grid, whereas we hope to attain the accuracy of the fine grid.

Figure 6.9 shows that in general the LGR predictions agree very well with the medium and fine grid predictions. Near the inlet section the LGR grid predictions are almost identical to the fine grid predictions, especially somewhat higher in the furnace. The predicted peak temperature is somewhat lower, although the medium and fine grid yield similar profiles. This might be contributed to the fact that the peak-temperature is a very local phenomena, and the coarser grid, further downstream, might still influence the results in the flamezone. Away from the symmetry plane the predictions are less satisfactory. In the lower part of the furnace the LGR grid systematically overpredicts the temperatures, and fails to reproduce the steep gradients. This stems from the fact that a relatively coarse grid is used in that region. However, further downstream (i.e near the outlet section the predictions agree very well with each other, although the measurements are less well reproduced. Note that near the outlet section the LGR grid is extremely coarse, and still able to perform correctly. This shows that the local grid refinement works as expected.
Figure 6.9: Comparison between simulations and in-flame measurements. $X$ is the distance from the symmetry plane, $Y$ is the distance from the inlet. Shown are the temperature profiles along the height of the furnace for medium grid (solid line), fine grid (dotted line) and locally refined grid (dashed line). Cubes are measurements.
6.5 Conclusions

In this chapter we demonstrated all the new features added in this study to perform the furnace simulations. We have shown that these feature have extended the realm of the code, as memory- and CPU-limits have been completely overcome. The visualisation now poses the most severe problem. We found that standard commercial visualisation tool are currently unable to handle large data sets efficiently.

The simulations agree well with experimental data. The steep temperature gradients are captured by the finer grids. The temperatures are slightly overpredicted near the inlet section, but this is caused by the non-inclusion of a soot model. The local grid refinement offers the flexibility we expected. The uniform underlying grid offers much more stable convergence behaviour. Near the outlet of the furnace, where the gradients are relatively small, a coarse grid is very capable of capturing the phenomena. Near the gas inlet the finer gridlevel render more accurate predictions. Some grid dependence is still found in the lower part of the furnace. Apparently finer grids are still needed in this region for accurate predictions, but with the local grid refinement demonstrated here, these regions can be found, and the distribution of nodes can really be fine tuned.

Also in terms of CPU-time, the use of LGR should be advocated. The runs with locally refined grids all converged with less problems and in less time than the runs without the local refinement. This a caused by the fact that less nodes are used in the computations, and the underlying grid is uniform.
7 Conclusions

7.1 Introduction

This chapter sums up the most important conclusions in this thesis. Every section treats a main objective as stated in section 1.3 of the introduction on page 5. For a more detailed derivation of these conclusions we refer to the conclusion sections at the end of each chapter, viz. sections 4.9, 5.8 and 6.5.

7.2 Numerical methods.

• The colocated and staggered variable arrangements yield similar predictions. Also the convergence behaviour does not differ much, if appropriate interpolation is used to avoid spurious pressure nodes. The colocated arrangement leads to easier programming, and is recommended for use in conjunction with multi-block multigrid computations.

• The SIP solver provides a good compromise between fast and efficient solvers for furnace simulations. It is recommended over GMRES and the SP-TDMA solver.

• Multigrid yields a convergence behaviour independent of the number of grid points. However, when the geometries contain non-orthogonal walls the porosity method has to be used to modify the Cartesian mesh. This hampers the performance of multigrid for furnace simulations.

7.3 Domain decomposition.

• Domain decomposition can be implemented in a general way to allow for easy parallelisation and local grid refinement for block-structured grids.

• The additive Schwartz method, which is suitable for parallelisation, is approximately twice as slow as the multiplicative Schwartz method. The exact number
depends on the accuracy of the solver within the blocks.

- Several algorithms have been implemented to update the values on the internal block-interfaces. The flux-conservation method converges faster than the straightforward update mechanism, because of the stronger coupling between blocks, but it quickly becomes unstable when more accurate solving of the systems is applied. Use of the straightforward update is recommended. Accurate solving for multi-block computations is not always the best way to obtain a numerically efficient code, because of the steep gradients arising on the interfaces.

- The convergence behaviour deteriorates when more blocks are used in a problem with a fixed size. The decrease in numerical efficiency was measured to be less than 10% in all cases considered.

- The decomposition of the domain should be done in such a way that the number of block-interfaces is as equal as possible in all directions (i.e. 16 blocks should be divided in $2 \times 2 \times 4$ rather than $1 \times 2 \times 8$ etc.). The most 'block-wise' decomposition, as opposed to the 'strip-wise' decomposition (say $1 \times 1 \times 16$) has a positive effect on the numerical efficiency and consequently on the CPU-time. This confirms similar observations of Brakkee (1996). This positive effect can be attributed to the fact that information can only travel one block per directions per boundary-update. The less blocks per direction there are, the less boundary-updates are needed for the information to travel through the entire domain.

- Increasing the number of iterations within the solver is not very useful and only leads to slightly improved convergence behaviour for multi-block simulations. This is caused by the fact that old values for the unknowns on the block-interfaces are used. Consequently, for improvement of the convergence behaviour of multi-block simulations, only a few iterations should be done within the solver; the values on the block interfaces should exchanged after these iterations, the matrix rebuilt and the source terms re-evaluated, and the new system with updated boundary conditions should be resolved again. This modified iteration loop significantly improves the convergence behaviour for multi-block simulations.

- Structured solvers offer very limited advantage for local grid refinement, as the refinement regions have to be mapped onto the blocks to save the banded structure of the system, which will lead to high load imbalance. At least locally unstructured grids should be applied.

- It is impossible to develop refinement criteria that give the desired local grid refinement without putting any pre-knowledge about the behaviour of the relevant quantities and their gradients.
7.4 Parallelisation.

- Domain decomposition is an easy way to port a sequential code to a parallel platform using message passing paradigms. Data parallel programming is currently not an option, since compilers are too immature.

- Proper cache use is very important if one wants to obtain good single PE performance on the CRAY T3E.

- The SPMD programming paradigm yields a well-maintainable code, with clear communication parts.

- Generic communication subroutines have been written to enable easy porting of the code to other platforms. These subroutines are general enough to be used in other codes too.

- Radiative heat transfer simulations using DTM can be very well parallelised.

- In our case the total time spent in actual communication was found to be less than 1% for all computations on the CRAY T3E. This is the results of the very fast network on this machine. Consequently, use of machine specific message passing libraries will not lead to a much faster code, as communication time is not the bottleneck in the code and so the use of portable libraries, such as MPI is advised.

- The most cumbersome part of the parallelisation is the load imbalance. This load imbalance can be only partially contributed to the fact that the load is not completely even on all PEs. Also the number of adjacent blocks influences the load imbalance.

- The performance of a code can be improved significantly if domains all have the same number of neighbours (up to 8 blocks), or if the updating could be done in the same time for all blocks. The size of the subdomains-interfaces does not influence the parallel performance if the latency and bandwidth are reasonable.

7.5 Furnace simulations

- The existing code has been extended with domain decomposition and is able to run on parallel platforms. The blocks can be locally refined.

- The principles of domain decomposition and parallelisation are well understood and have been used successfully to perform large simulations on complex flows, namely turbulent combustion in gas-fired glass melting furnaces.
The restrictions with respect to CPU-time and memory have been completely overcome in this study. The grids can have arbitrary size and can be run in reasonable turn-around time. We have performed simulations with up to 2 million grid points, which is 12 times larger than grids used in previous studies of Boerstoel (1997).

The simulations of the IFRF-furnace agree well with previous simulations by Boerstoel (1997), except from some small expected differences which are caused by the use of colocated grids and consequently different implementation of the boundary conditions.

The agreement of the simulations with the IFRF-furnace experiments are satisfactory too, especially in the upper part of the furnace, in the recirculation zone. The predictions will improve with the re-insertion of the soot-model, the use of improved turbulence models and more accurate radiation models. The current study shows that use of these models has come within the realm of computation.

The local grid refinement offers a good tool to analyse grid-dependence in certain parts of the flow and to fine tune the distribution of the nodes. For the IFRF-furnace simulations the same accuracy was obtained with less points than without the local grid refinement.
Appendix
A Communication subroutines for parallel computing

This appendix contains the fortran90 source of the generic subroutines for parallel computing. The error checks, inclusion of common blocks and the declaration of local variables are removed for reasons of clarity.

A.1 Subroutine parstart

SUBROUTINE parstart
!=============================================================================
! Set up the parallel program. In the static set-up, the
! group-size determines the number of blocks, since the mapping
! is bijective.
!=============================================================================
! ... Use Statements ...
USE mpi_module
!
! ... External Subroutines ...
EXTERNAL mpi_comm_rank, mpi_comm_size, mpi_init
!
! Enroll task under MPI
!
CALL mpi_init(info)
CALL mpi_comm_rank(mpi_comm_world,myproc,info)
CALL mpi_comm_size(mpi_comm_world,nproc,info)
!
! Perform error checks
!
iproc = 0
IF (myproc<0) THEN
   WRITE (*,*) 'ERROR: Parallel program not started correctly.'
iproc = 1
END IF
CALL parstop('test','parstart',iproc)
END SUBROUTINE parstart

143
A.2 Subroutine parstop

SUBROUTINE parstop(action, procname, ierr)
!
! Test if program should be terminated.
! If action = 'stop', then program ends with normal message
! If action = 'error', then program ends with error message.
! If action = 'test', then program only ends if the sum of all
! errors > 0, otherwise, just continues
!
! i action : see above
! i procname : name of the calling procedure
! i error : integer defining whether this block wants
! to quit (>0) or not (=0)
!
! .. Use Statements ..
USE mpi_module
!
! .. Arguments ..
INTEGER :: ierr
CHARACTER (*) :: action, procname
!
! .. External Subroutines ..
EXTERNAL mpi_allreduce, mpi_finalize
!
! .. Intrinsic Functions ..
INTRINSIC trim
!
! Normal termination
!
IF (trim(action)=='stop') THEN
  IF (myproc==master) THEN
    CALL writeoutputheader('Normal termination of the program')
    WRITE (iout, '(2A/)') 'Furnace terminated from procedure ', &
    trim(procname)
  END IF
END IF
!
! Error termination
!
ELSE IF (trim(action)=='error') THEN
  IF (myproc==master) THEN
    CALL writeoutputheader('Furnace program crashed')
    WRITE (iout, '(2A/)') 'Furnace terminated from procedure ', &
    trim(procname)
  END IF
END IF
!
! Test if termination
!
ELSE IF (trim(action)=='test') THEN
  CALL mpi_allreduce(ierr2 ierr2,1,mpi_integer,mpi_sum, &
                          mpi_comm_world,info)
  IF (ierr2==0) RETURN
  IF (myproc==master) THEN
    CALL writeoutputheader('Furnace program crashed during test')
    WRITE (iout,*) 'ERROR: ', ierr2, ' PE(s) crashed ', &
    'Entire program terminated in ', trim(procname)
  END IF
END IF
!
! Illegal test value.

ELSE
  IF (myproc==master) THEN
    WRITE (iout,*) 
    'ERROR: Action should be <test>, <stop>, <error>.'
    CALL writeoutputheader('Furnace program crashed')
    WRITE (iout,'(A)') &
    'Furnace terminated from procedure parstop'
  END IF
END IF

! Stop the program
! ------------------------------------------------------
IF (myproc==master) THEN
  CALL timestamp('Program stopping time')
  CALL writeoutputheader('End of Furnace output file')
  CLOSE (iout)
END IF
CALL mpi_finalize(info)
STOP 'Program terminated normally'
END SUBROUTINE parstop

A.3 Subroutine parsend

SUBROUTINE parsend(rarray,iarray,carray,block,request)
! ------------------------------------------------------
! This subroutine sends an array to one specific other processor
! i rarray    : REAL    : real array to be sent
! i iarray    : INTEGER : Integer array to be sent
! i carray    : CHAR    : string to be sent
! i block    : INTEGER  : Number of the block to which the
!                       : data must be sent. Processor number
!                       : is then given in procnr-array
! ! .. Use Statements ..
USE mpi_module

! .. Scalar Arguments ..
INTEGER, INTENT (IN) :: block
INTEGER, INTENT (OUT) :: request
CHARACTER (*) , OPTIONAL, INTENT (INOUT) :: carray
REAL (kind=ep) , OPTIONAL, INTENT (INOUT), DIMENSION (:) :: rarray
INTEGER, OPTIONAL, INTENT (INOUT), DIMENSION (:) :: iarray

! .. External Subroutines ..
EXTERNAL mpi_isend

! .. Intrinsic Functions ..
INTRINSIC int, len, present, size, sum

! ------------------------------------------------------
! Error checks
! ------------------------------------------------------
! Exactly 1 array can be sent!
...

! Check for valid processor number
...

! ------------------------------------------------------
! Get length
A.4 Subroutine parrecv

SUBROUTINE parrecv(array, iarray, carray, block)
! This subroutine receives an array from one specific other processor
! i array : REAL  : real array to be sent
! i array : INTEGER : Integer array to be sent
! i array : CHAR  : string to be sent
! i block : INTEGER : Number of the block from which to receive
! Processor number determined from proco.
!
USE mpi_module
!
! Arguments ...
INTEGER, INTENT (IN) :: block
CHARACTER (*) , OPTIONAL, INTENT (INOUT) :: carray
REAL (kind=wp) , OPTIONAL, INTENT (INOUT), DIMENSION (: ) :: rarray
INTEGER, OPTIONAL, INTENT (INOUT), DIMENSION (: ) :: iarray
!
! Local ...
INTEGER, DIMENSION (mpi_status) :: status
!
! External Subroutines ...
EXTERNAL mpi_recev
!
! Intrinsic Functions ...
INTRINSIC int, len, present, size, sum
!
! Error checks
!
! Exactly 1 array can be sent!
!
! Check for valid processor number
!
! Get length
A.5. Subroutine parrecv

SUBROUTINE parrecv(oper, rarray, iarray, carray, root)

! Performs the all2one, one2all and reduce operations. This yields
! a more effective communication than using point2point only.
!
! variables in the argument list
! i oper : Global operation to be performed.
! ! Currently implemented:
! all2one : All blocks send to 'root'
! one2all : 'root' sends to all blocks
! global_sum : All blocks add values in [r/i]array,
! global_max : Dito, but maximum ...
! barrier : Global synchronisation
! i/o rarray : Real array to be sent
! i/o iarray : Integer array to be sent
! i/o carray : String array to be sent
! i root : Root block for all2one and one2all.
! ! Not needed otherwise
!
! USE mpi_module
!
! Arguments ...
INTEGER, INTENT (IN) :: oper
INTEGER, OPTIONAL, INTENT (IN) :: root
CHARACTER (*) , OPTIONAL, INTENT (INOUT) :: carray
REAL (kind=wp), OPTIONAL, INTENT (INOUT), DIMENSION (::) :: rarray
INTEGER, OPTIONAL, INTENT (INOUT), DIMENSION (::) :: iarray
!
! External Subroutines ...
EXTERNAL mpi_allreduce, mpi_barrier, mpi_bcast, mpi_gather
!
! Intrinsic Functions ...
INTRINSIC int, len, present, size
!
! Error checks

! Receiving the message using a blocking receive
!
! IF (present(carray)) THEN
! CALL mpi_recv(carray, length, mpi_character, &
!               procmo(block), mpi_any_tag, &
!               mpi_comm_world, status, info)
! ELSE IF (present(rarray)) THEN
! CALL mpi_recv(rarray, length, mpi_double_precision, &
!               procmo(block), mpi_any_tag, &
!               mpi_comm_world, status, info)
! ELSE IF (present(iarray)) THEN
! CALL mpi_recv(iarray, length, mpi_integer, &
!               procmo(block), mpi_any_tag, &
!               mpi_comm_world, status, info)
! END IF
!
END SUBROUTINE parrecv
Exactly 1 array can be sent!

Get length

IF (present(array)) length = int(size(array))
IF (present(array)) length = int(size(array))
IF (present(array)) length = len(array)

Perform operation

IF (op == one2all) THEN

Broadcast to all

IF (present(array)) THEN
    CALL mpi_bcast(array,length,mpi_character, &
                  prochno(root), mpi_comm_world,info)
ELSE IF (present(array)) THEN
    CALL mpi_bcast(array,length,mpi_double_precision, &
                  prochno(root), mpi_comm_world,info)
ELSE IF (present(array)) THEN
    CALL mpi_bcast(array,length,mpi_integer, &
                  prochno(root), mpi_comm_world,info)
END IF

ELSE IF (op == all2one) THEN

Gather from all

IF (present(array)) THEN
    CALL mpi_gather(array,length,mpi_character, &
                   array,length,mpi_character, &
                   prochno(root),mpi_comm_world,info)
ELSE IF (present(array)) THEN
    CALL mpi_gather(array,length,mpi_double_precision, &
                   array,length,mpi_double_precision, &
                   prochno(root),mpi_comm_world,info)
ELSE IF (present(array)) THEN
    CALL mpi_gather(array,length,mpi_integer, &
                   array,length,mpi_integer, &
                   prochno(root),mpi_comm_world,info)
END IF

ELSE IF (op == global_sum) THEN

Global sum

IF (present(array)) THEN
    CALL mpi_allreduce(array,array,length, &
                       mpi_double_precision, mpi_sum, &
                       mpi_comm_world,info)
ELSE IF (present(array)) THEN
    CALL mpi_allreduce(array,array,length, &
                       mpi_integer,mpi_sum, &
                       mpi_comm_world,info)
ELSE
    IF (myproc == master) THEN
        WRITE (iuout,'(a)') 'ERROR: Characters found for global sum.'
ELSE IF (oper==global_max) THEN
  ------------------------
  Global maximum
  ------------------------
  IF (present(rarray)) THEN
    CALL mpi_allreduce(rarray, rarray, length, &
                      mpi_double_precision, mpi_max, &
                      mpi_comm_world, info)
  ELSE IF (present(iarray)) THEN
    CALL mpi_allreduce(iarray, iarray, length, &
                      mpi_integer, mpi_max, &
                      mpi_comm_world, info)
  ELSE
    IF (myproc==master) THEN
      WRITE (iout,*), 'ERROR: Characters found for global max.'
    END IF
  END IF
  CALL parstop('error','parglobal',0)
END IF
ELSE IF (oper==barrier) THEN
  ------------------------
  Barrier
  ------------------------
  CALL mpi_barrier(mpi_comm_world, info)
ELSE
  ------------------------
  Undefined global operation
  ------------------------
  IF (myproc==master) THEN
    WRITE (iout,*), 'ERROR: Operation ',oper,' not implemented.'
  END IF
  CALL parstop('error','parglobal',0)
END IF
END SUBROUTINE parglobal
Notations

Roman symbols

\( a \) coefficient in discretization (m\(^2\))
\( A \) area cell wall
\( b \) source term in discretization
\( C_\mu \) empirical constant \( k-\varepsilon \) model (0.09) (-)
\( C_{\varepsilon,1} \) empirical constant \( k-\varepsilon \) model (1.44) (-)
\( C_{\varepsilon,2} \) empirical constant \( k-\varepsilon \) model (1.92) (-)
\( c_p \) specific heat at constant pressure (J kg\(^{-1}\) K\(^{-1}\))
\( c_{p,i} \) specific heat at constant pressure for species \( i \) (J kg\(^{-1}\) K\(^{-1}\))
\( E \) Constant in law of the wall (9.8) (-)
\( E \) Activation energy (J kg\(^{-1}\))
\( E_{\text{total}} \) Total energy of the flow (J kg\(^{-1}\))
\( f \) mixture fraction (-)
\( g \) mixture fraction fluctuation (-)
\( g_i \) gravitational acceleration (m s\(^{-2}\))
\( i' \) radiative intensity (W m\(^{-2}\) sr\(^{-1}\))
\( J \) sum of convective and diffusive flux
\( h \) enthalpy per unit mass (J kg\(^{-1}\))
\( k \) turbulent kinetic energy (m\(^2\) s\(^{-2}\))
\( M_i \) Molair mass fraction of species \( i \) (kg kmole\(^{-1}\))
\( p \) pressure (Pa)
\( P_k \) Production of turbulent kinetic energy (kg m\(^{-1}\) s\(^{-3}\))
\( Pe \) Peclet number (-)
\( q \) radiative heat flux (W m\(^{-2}\))
\( R^0 \) universal gas constant (J kmole\(^{-1}\) K\(^{-1}\))
\( S \) source term in transport equation
\( T \) temperature (K)
\( t \) time (s)
\( u_j \) velocity component in \( j \)-direction (m s\(^{-1}\))
\( w_f \) Reaction rate (kg m\(^{-3}\) s\(^{-1}\))
\( x_j \) coordinate component in \( j \)-direction (m)
\( Y_i \) Mass fraction of species \( i \) (-)

Greek symbols

\( \Gamma \) effective transport coefficient (kg m\(^{-1}\) s\(^{-1}\))
\( \delta_{ij} \) Kronecker delta function (-)
\( \Delta V \) volume of control volume (m\(^3\))
\( \varepsilon \) dissipation rate of turbulent kinetic energy (m\(^2\) s\(^{-3}\))
\( \kappa \) von Karman’s constant (0.42) (-)
\( \lambda \) thermal conductivity (W m\(^{-1}\) K\(^{-1}\))
### Notations

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Unit(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu$</td>
<td>dynamic viscosity</td>
<td>$\text{kg m}^{-1} \text{s}^{-1}$</td>
</tr>
<tr>
<td>$\mu_t$</td>
<td>turbulent viscosity</td>
<td>$\text{kg m}^{-1} \text{s}^{-1}$</td>
</tr>
<tr>
<td>$\rho$</td>
<td>density</td>
<td>$\text{kg m}^{-3}$</td>
</tr>
<tr>
<td>$\tau$</td>
<td>transmissivity</td>
<td>$\text{(-)}$</td>
</tr>
<tr>
<td>$\tau_{ij}$</td>
<td>shear stress tensor</td>
<td>$\text{Pa}$</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>turbulent Prandtl or Schmidt number</td>
<td>$\text{(-)}$</td>
</tr>
<tr>
<td>$\phi$</td>
<td>general variable</td>
<td>$\text{(-)}$</td>
</tr>
</tbody>
</table>

### Abbreviations

- **CFD**: Computational Fluid Dynamics
- **CDS**: Central Differencing Scheme
- **CPU**: Central Processing Unit (Heart of the computer)
- **CPU-time**: Time actually spent by a computer to perform the computation
- **CV**: Control Volume
- **DTM**: Discrete Transfer Method
- **FMG**: Full MultiGrid
- **FDM**: Finite Difference Method (Discretisation method)
- **FEM**: Finite Element Method (Discretisation method)
- **FVM**: Finite Volume Method (Discretisation method)
- **(M)FLOPS**: (Mega) FLOating Point Operations per Second
- **GS**: Gauss-Seidel (Solver)
- **GMRES**: Generalised Minimal RESidual algorithm (Solver)
- **HPF**: High Performance Fortran (Data-parallel programming language)
- **IC-CG**: Incomplete Choleski Conjugate Gradients (Solver)
- **IC-CGSTAB**: Incomplete Choleski Conjugate Gradients STABilised(Solver)
- **IFRF**: International Flame Research Foundation
- **LGR**: Local Grid Refinement
- **MPI**: Message Passing Interface (Message passing protocol)
- **PVM**: Parallel Virtual Machine (Message passing protocol)
- **PE**: Processing Element (computational node)
- **RHS**: Right Hand Side
- **RISC**: Reduced Instruction Set Computer (chip architecture)
- **SIMPLE**: Semi-Implicit Method for Pressure-Linked Equations (hydrodynamical coupling)
- **SIP**: Strongly Implicit Procedure (Solver)
- **SPMD**: Single Program, Multiple Data (Programming paradigm)
- **SP-TDMA**: Space-Tri-Diagonal-Matrix-Algorithm (Solver)
- **TVD**: Total Variation Diminishing (convection scheme)
- **UDS**: Upwind Differencing Scheme
- **(N)UMA**: (Non) Uniform Memory Access

Wall-Clock time Total elapsed time during the program execution.
Bibliography


BIBLIOGRAPHY


Stellingen

1. Domein decompositie is een makkelijke manier om sequentiële codes over te zetten naar parallele codes, met gebruik van 'message passing' paradigma's. 'Data parallel' programmeren is momenteel nog geen goede keuze, aangezien compilers nog niet ontwikkeld genoeg zijn.


3. De locale roosterverfijning is een krachtige manier om grid-onafhankelijkheid te analyseren in delen van het rekendomein, en om de verdeling van roosterpunten over het domein gedetailleerd af te stellen. Voor een gegeven nauwkeurigheid zijn er met locale roosterverfijning minder roosterpunten nodig dan met globale roosterverfijning.

4. Parallel rekenen is de technologie van de toekomst en zal dat, bij de huidige voortgang in compilers, nog een tijd blijven.

5. Zowel de promotendis als de universiteit zijn gebaat bij een exactere en haalbaar definitie van de doelen van een promovendus. Meer duidelijkheid omtrent de uitkomst van onderzoek kan tot een betere interactie in de groep leiden.

6. Het is moeilijk niet te worden wat je niet wilt worden.

7. Een verassingsmenu is nooit uitverkocht.

8. Het is voor de geloofwaardigheid van een Nederlandse salsa band aan te raden tenminste één persoon in de band te hebben die er Afro-cubaans uitziet.

9. Tolerantie wordt vaak verward met apathie.

10. Er is geen verband tussen de complexiteit van een muziekstuk en het succes ervan. Dit geldt wel voor televisie programma's.

11. We staan te vroeg op omdat we teveel te doen hebben. We gaan te vroeg naar bed omdat te weinig om over te praten hebben. (Oscar Wilde-Letters)

12. Gelukkig kunt je niet worden, alleen zijn.

13. Opscheppen over eigen (vermeende) kwaliteiten impliceert dat de persoon die kwaliteiten (nog) niet bezit.

14. Delft is leuker als je in de binnenstad woont.

15. Een individualistischer samenleving begint bij jezelf.

160
Propositions

1. Domain decomposition is an easy way to port a sequential code to a parallel platform using message passing paradigms. Data parallel programming is currently not an option, since compilers are too immature.

2. The collocated and staggered grids yield similar predictions, if appropriate interpolation is used to avoid spurious pressure nodes. Also the convergence behaviour does not differ much in both approaches. The collocated grids lead to easier programming, and are recommended for use in multi-block multigrid computations.

3. The local grid refinement offers a powerful tool to analyse grid-independence in certain parts of the flow and to finetune the distribution of the nodes. For a given accuracy the local grid refinement requires less nodes than the global grid refinement method.

4. Parallel computing is the technology of the future and always will be, with the current progress in compilers.

5. Both promovendi and the university benefit from a more specified and achievable research goal of the promovendus. More clarity about the outcome of research can lead to an improved interaction in the group.

6. It is difficult not to become what you don't want to become.

7. A surprisemenu is never sold out.

8. For the credibility of a Dutch salsa-band it is advised to have at least one person in the band with Afro-Cuban looks.

9. Tolerance is often confused with apathy.

10. There is no correlation between the complexity of a piece of music and its success. This correlation does exist for television programs.

11. We get up too early because we have too much to do. We go to bed too early because we have too little to talk about. (Oscar Wilde-Letters).

12. Happy is something you cannot become, only be.

13. To brag about (assumed) qualities implies that the person does not (yet) have those qualities.

14. Delft is more fun if you live in the inner city.

15. A more individualistic society starts with yourself.
Curriculum vitae

Personal
Date of birth 22 August 1970
Place of birth Gouda, The Netherlands
Nationality Dutch

Education
1994–1998 Ph.D. Research
Delft University of Technology, Department of Applied Physics.

Delft University of Technology, Department of Technical Mathematics. Graduation project: Improving the numerical prediction methods for fluid flow around ship hulls. This project concerned the design and implementation of novel prediction methods for numerical simulations of flows in complex geometries.

1982–1988 Secondary school
Grammar school 'gymnasium', Gouda, The Netherlands.

Work experience
1999– Math/physics teacher for VSO in Cameroon, Africa.
1998–1999 Post Doctoral position at TNO-TPD.
1996 3 months combined training and research at the Edinburgh Parallel Computing Centre, Edinburgh, Scotland.